THE GENESIS PAPERS: VOLUME 1

Don S. Batory

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-20 June 1991

ABSTRACT

This is a collection of major publications on the Genesis extensible DBMS project. There are six papers overall. Three explain the Genesis open-system architecture concept, two papers discuss its implementation, and the last paper is the Genesis Unix User's Manual.

Keywords: Genesis, extensible DBMSs, open system architectures, software reuse.

The Genesis Papers: Volume 1

Genesis is the first software building-blocks technology for constructing database management systems. The Genesis 2.0 prototype shows that complex and customized DBMSs can be assembled literally in minutes from prefabricated components. As new components can be added to existing libraries, Genesis is an extensible open-system architecture for DBMSs.

This technical report collects together major publications on Genesis. The first three lay the theoretical foundations for a software building-blocks technology. Paper [1] gave the first indication that differences among commercial DBMSs could be explained by different compositions of standardized components. Paper [2] formalized these ideas by recognizing the parametric nature of components, and that DBMSs could be modeled as type expressions. The realization that these concepts were not limited to DBMSs and could be applied to other domains with large software systems is presented in Paper [3].

The next two papers discuss implementation issues. The kernel of Genesis, called Jupiter, presented file structure and database recovery capabilities. The implementation of Jupiter is reviewed in Paper [4]. Paper [5] presents an overview of the Genesis layout editor, called DaTE. It turns out that no all compositions of components are meaningful. DaTE embodies sophisticated design rule checking to ensure that systems specified through DaTE will be operational. The design rule checking algorithms are presented in [5].

Paper [6] is the Unix User's Manual for Genesis 2.0. Explanations of how DBMSs are specified through compositions of components are presented, along with a summary of currently available features.

[1] D. S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', **ACM Transactions on Database Systems**, 10,4 (Dec. 1985).

[2] D. S. Batory, 'Concepts for a Database System Synthesizer', **ACM PODS 1988**.

[3] D. S. Batory and S. W. O'Malley, 'A Definition of Open Architecture Systems with Reusable Components: Preliminary Draft', **Proc. Domain Modeling Workshop**, 1991.

[4] D. S. Batory, J. R. Barnett, J. Roy, B. C. Twichell, and J. Garza, 'Construction of File Management Systems from Software Components', **Proc. COMPSAC 1989**.

[5] D. S. Batory and J. R. Barnett, 'DaTE: The Genesis DBMS Software Layout Editor', **Conceptual Modeling, Databases, and CASE**, R. Zicari, ed., MacGraw-Hill, 1991.

[6] D. S. Batory, 'The Genesis Unix Users Manual', University of Texas Tech. Rep. TR-90-38, 1990.

Modeling the Storage Architectures of Commercial Database Systems

D. S. BATORY
The University of Texas at Austin

Modeling the storage structures of a DBMS is a prerequisite to understanding and optimizing database performance. Previously, such modeling was very difficult because the fundamental role of conceptual-to-internal mappings in DBMS implementations went unrecognized.

In this paper we present a model of physical databases, called the *transformation model*, that makes conceptual-to-internal mappings explicit. By exposing such mappings, we show that it is possible to model the storage architectures (i.e., the storage structures and mappings) of many commercial DBMSs in a precise, systematic, and comprehendible way. Models of the INQUIRE, ADABAS, and SYSTEM 2000 storage architectures are presented as examples of the model's utility.

We believe the transformation model helps bridge the gap between physical database theory and practice. It also reveals the possibility of a technology to automate the development of physical database software.

Categories and Subject Descriptors: E.5 [**Data**]: Files—*organization/structure*; H.2.2 [**Database Management**]: Physical Design—*access methods*

General Terms: Design Documentation

1. INTRODUCTION

Optimizing the performance of commercial database systems is a significant and very difficult problem. Progress toward its solution has come from models of *physical databases* (i.e., models of database storage structures and their associated search and maintenance algorithms). Since 1970 there have been important advances in file structure and physical database modeling. These advances, as a rule, have been incorporated into a progression of increasingly more sophisticated and realistic general-purpose models [6, 29, 41, 55, 67, 68, 88].

In spite of progress, there still is no model that can account for the diversity and complexity of storage structures and algorithms found in commercial DBMSs in a comprehendible way. Although some models have been used as starting points, considerable effort is needed to adapt and extend them just to describe a single DBMS ([15]). In view of these difficulties, it is easy to understand why there are so few design and performance aids for commercial systems ([33, 34]).

This work was supported by the National Science Foundation under grant MCS-8317353 and the U.S. Department of Energy under contract DE-A505-81ER10977.

The problems in using current models clearly indicate that some fundamental principles of database system implementation are not well understood. A careful examination of several commercial DBMSs reveals that current models presume conceptual-to-internal mappings are simple. That is, given a set of conceptual files and links, there is an obvious mapping to their internal counterparts. In almost all commercial and specialized DBMSs this is definitely not the case.

In this paper we present the *transformation model* (TM), a model of physical databases that makes conceptual-to-internal mappings explicit. We identify a set of primitive mappings, called *elementary transformations*, and show how compositions of these transformations can be used to accurately express the storage architectures of operational DBMSs. By *storage architecture* we mean the combination of conceptual-to-internal mappings, file structures, and record-linking mechanisms that define a physical database. As examples of the TM's practicality, we show how the diverse and complex storage architectures that underlie three commercial DBMSs—namely, INQUIRE, ADABAS, and SYSTEM 2000—can be defined in a precise, systematic, and simple way. Models of other commercial DBMSs—relational (MRS, INGRES), network (IDMS, DMS-1100), and statistical (RAPID, ALDS, CREATABASE)—are presented in [7, 8] and [9]. A preliminary model of IMS has also been developed ([8]).

A primary goal of this paper is to explain conceptual-to-internal mappings of data. Mappings of operations (e.g., record retrieval, insertion, deletion) are discussed only briefly, but are considered in more detail in [9] and [86].

We believe our research makes four main contributions: (1) it is a step toward the development of practical design and tuning aids for operational DBMSs; (2) it provides a basis for a technology to automate the development of physical database software; (3) it introduces practical tools to design, communicate, and understand prototype storage architectures; and (4) it signals the beginning of a comprehensive reference to the storage architectures of commercial DBMSs. These and other contributions are discussed in Section 5.

The starting point of our research is the Unifying Model (UM) of Batory and Gotlieb [6]. In the following section we review the basic concepts of the UM and its subsequent extensions. We explain in Appendix I how these extensions subsume earlier studies, thereby establishing the UM as a framework in which most, if not all, contributions to file and physical database research may eventually be cast. Special attention is given to show how the UM can be reduced to DIAM ([29, 67]). In the following section we present an example which clearly reveals the limitations of the generalized UM (and its predecessors) and motivates the study of conceptual-to-internal mappings.

2. THE UNIFYING MODEL: A GENERALIZATION

The UM was shown to relate and extend disparate works on file design and optimization, transposed files, batched searching, index selection, and file reorganization, among others. However, the UM could not account for certain classes of storage structures (e.g., clustering and hierarchical sequential record linkages) that are commonly found in commercial DBMSs. Nor did it distinguish between the logical concepts of files and links and their physical implementations (i.e., simple files and linksets). In the following paragraphs we explain a generalization of the UM framework that makes these important distinctions and accommodates these structures. Additional details are presented in Appendix II.

Physical databases can be *decomposed* into a collection of internal files and internal links. An *internal file* is a set of records that are instances of a single record type. A relationship between two or more internal files is an *internal link*. Internal links can be understood as generalizations of CODASYL sets; each internal link relates records of one file, called the *parent file*, to records of other files, called *child files*. (We draw a distinction here between conceptual files and links, which are defined in database schemas, from internal files and internal links. We will see later that they are quite different.)

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 hierarchical sequential lists. Linksets also deal with the clustering of child records around their parent records (i.e., sequential placement or [24] or "store near" [22]). Catalogs of recognized simple files and linksets are given in Appendix II.

The structure of a physical database can therefore be specified by (1) decomposing the database into its internal files and links and (2) assigning each internal file to a simple file structure and each internal link to a linkset structure. Classical examples of decomposition are presented in the next section, along with the introduction of notation which will be used extensively later.

2.1 Examples: Decomposition of Inverted and Multilist Files

Inverted and multilist files are classical file structures, but they are not simple file structures. Rather, they are actually networks on interconnected files that have special implementation connotations.

Consider a file of records of type DATA. Suppose DATA records are stored in an inverted file where attributes F_j and F_k are indexed. The first step in defining the implementation of the inverted file is to decompose it. Decomposition reveals three internal files and two internal links. One file is the DATA file; the other two are INDEX$_j$ and INDEX$_k$, one file for each of the indexed attributes. Each INDEX file is connected to the DATA file by precisely one link, where the INDEX file assumes the role of parent. Relationships between files and links are shown graphically in a *data structure diagram* (dsd) where boxes represent files and arrows denote links. (Arrows are drawn from parent files to their child files). Figure 1.dsd (abbreviation for the dsd of Figure 1) shows the decomposition of the inverted file. The remaining parts of Figure 1 are explained in the next section.

The second step is to assign implementations to the internal files and links. A common assignment has each INDEX file organized by a distinct B+ tree, and the DATA file organized by an unordered file structure. Thus, there is a total of three simple files (i.e., a DATA file structure and two INDEX file structures). The internal links would be implemented by inverted lists or pointer arrays.

Note that other simple file assignments are possible. For example, one INDEX file could be implemented by an unordered file, the other by an indexed-sequential

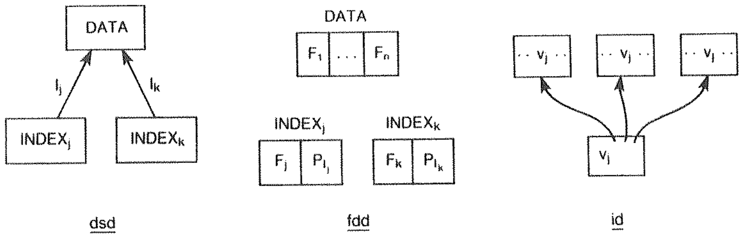


Fig. 1. Decomposition of an inverted file.

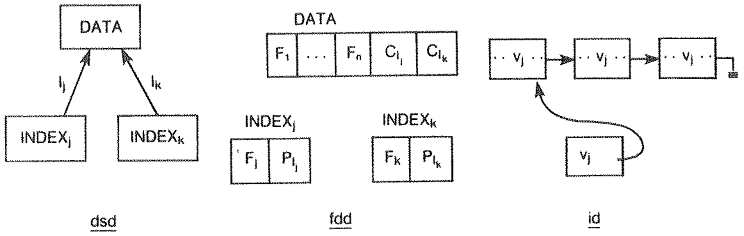


Fig. 2. Decomposition of a multilist file.

file, and the DATA file might be stored in a hash-based file. Such generalizations follow naturally from decomposition. (INGRES, incidentally, is based on inverted files and allows such implementation possibilities [76]).

Now consider another example. Suppose DATA records are stored in 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 (Figure 2.dsd). Furthermore, typical multilist file implementations are quite similar to inverted file implementations: each INDEX file is organized by a distinct B+ tree, and the DATA file is organized by an unordered file structure. However, the link implementations are different: multilist files use multilist (i.e., list) linksets.

It is worth noting that the INDEX files of inverted and multilist files correspond to secondary indices. The term *primary index* has been used by some researchers to mean the indexing structure that directs the clustering of internal records on their primary key. We prefer to use the term *cluster index* instead, since a cluster index is actually part of a simple file structure, as opposed to being a distinct file as is the case with secondary indices. In the UM, every simple file is a combination of a cluster index and an internal record storage structure, called the *data level*. Thus, B+ trees, indexed-sequential, dynamic hash-based structures, etc., all have a clearly identifiable cluster index and data level. This means in the above examples that a primary index (cluster index) is provided automatically to each INDEX file and DATA file by virtue of being organized by a simple file structure. It is in this way that the UM handles the concept of primary indices.

Implementations of physical databases can be described in further detail by introducing additional diagrammatic notations and by extending the conventions of data structure diagrams to express *N*:*M* links. This is done in the following section.

2.2 Additional Background

Two other diagrams are useful in elaborating implementation details of physical databases. One is a *field definition diagram* (fdd), which shows the fields of the record types that appear in a data structure diagram. Consider again the inverted file of Figure 1. Figure 1.fdd (abbreviation for the fdd of Figure 1) shows the DATA record type to consist of fields $F_1 \ldots F_n$. 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}. We refer to P_{I_j} as the *parent field* of linkset I_j. The INDEX$_k$ type has a format similar to INDEX$_j$.

The other diagram is an *instance diagram* (id), which is used to illustrate the implementation of one or more link occurrences. A *link occurrence* consists of a single parent record and the zero or more child records to which it is related. Instance diagrams serve to further elaborate data structure and field definition diagrams. To minimize the clutter in instance diagrams, records are not labeled with their types. Instead the types can be inferred by their positions or contents relative to the associated fdd or dsd. Figure 1.id (abbreviation for the id of Figure 1) shows the implementation of an I_j link occurrence. An INDEX$_j$ record is shown containing data value v_j and an inverted list which references all DATA records (three are shown) that have v_j as their F_j value. An instance diagram of an I_k link occurrence implementation would be drawn identically to that of Figure 1.id, except for the labeling (v_k would be used to denote a F_k value). In cases such as this, where instance diagrams would be duplicated, we show only one.

The field definition and instance diagrams for the multilist file are shown in Figures 2.fdd and 2.id. Note that 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 linksets I_j and I_k. Their purpose is to contain pointers to the next DATA record on a list of DATA records. Figure 2.id shows the same link occurrence of Figure 1.id, except that a list structure connects an INDEX$_j$ record to its DATA records.

We use the terms *parent field* and *child field* as generic names to refer to fields that must be present in parent and child records, respectively, in order to realize particular linkset structures. Some parent and child fields have common names, such as inverted list fields and parent pointer fields. But most do not. Another reason for their use is that they define semantically meaningful fields whose contents can be quite complex. The parent field of an inverted list, for example, not only contains an array of pointers, but also a count subfield which contains the number of pointers in the array and possibly the length of the array in bytes. By treating parent and child fields atomically, implementation details of linksets that are irrelevant to understanding storage architectures can be hidden.

As a general rule, the presence and function of parent and child fields in record types that are linked is determined solely by the underlying linkset. In the case of inverted list linksets (Fig. 1), a parent field appears in every parent record.

For multilists (Fig. 2), both parent and child fields are used. IMS logical parent pointers are linksets that are implemented solely by parent pointers [24]; 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.

The pointer structures, count fields, and so on that are present in parent fields are usually different than those found in child fields. As a consequence, linksets have a directionality (i.e., parent and child files of a linkset must be distinguished in order to determine the placement of the parent and child fields of the linkset). Links, in contrast, express logical relationships which do not have a directionality. Thus, the directionality of links (arrows) in data structure diagrams serve to indicate the roles files play in link implementations.

Assigning the directionality of links in data structure diagrams is quite simple. Most linksets implement 1:N links. In the tradition of the CODASYL model, 1:N links are represented by arrows drawn in the direction of the "N" part of the relationship; the file at the "1" side is the parent and those at the "N" side are the children. We follow this tradition. However, links can also express 1:1 and M:N relationships. Usually, the linksets that implement these links are obvious generalizations or specializations of 1:N linksets, so a directionality can be assigned as in the 1:N case. We encounter an example of this (M:N multilists) in our discussion of INQUIRE in Section 4. When neither child or parent fields are introduced by a linkset or when no distinction between parent and child files can be made, bidirectional links ($A \longleftrightarrow B$) which do not force parent and child distinctions may be used. Examples of bidirectional links arise in our discussions of transposition and actualization in Section 3, and the couplings of ADABAS in Appendix III.

In Appendix I we explain how all of the major general-purpose models of physical databases that predated the UM are subsumed by this framework. Even so, this framework is still inadequate to model the storage architectures of operational DBMSs. Correcting the problem does not simply involve enlarging the spectrum of structures and operations the UM describes. It requires much more. The next section illustrates the limitations of this framework.

2.3 Limitations of Current Models

Consider the inverted file of Figure 3, which has a single INDEX file that inverts field F. INDEX records are obviously variable-length. But suppose that the file structures that underlie the inverted file can only handle fixed-length records. How can variable-length INDEX records be stored?

A common solution (one of many possible) is to divide INDEX records into one or more fixed-length fragments. The first fragment, here called a PRIMARY record, contains the data field F and a number of pointers. The other fragments are SECONDARY records (sometimes called overflow records), and they contain the remaining pointers. PRIMARY and SECONDARY records are connected by link L. L is usually implemented as a list linkset.

Figure 4 illustrates this solution using some notation and relationships that are explained in a more comprehensive setting in Section 3. In Figure 4.dsd, the dashed outline of the INDEX file indicates that an INDEX record is mapped to a PRIMARY record and zero or more SECONDARY records connected via link L. $\div P_I$ in Figure 4.fdd is the name of the field (in both PRIMARY and SECONDARY) that contains a fragment of the contents of field P_I. Figure 4.id shows how the INDEX record of Figure 3.id was divided into four fragments: one PRIMARY and three SECONDARY. SECONDARY records are connected to PRIMARY records by a list linkset. In this example both PRIMARY and SECONDARY records contain pointers (part of P_I) to DATA records.

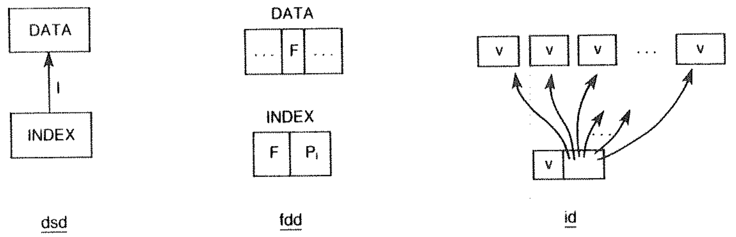


Fig. 3. An inverted file with one index file.

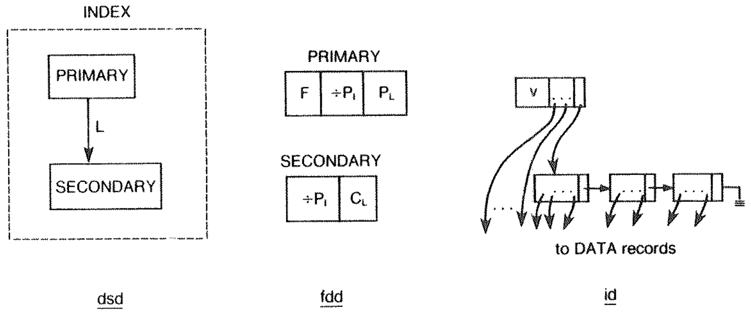


Fig. 4. Mapping of variable-length INDEX records to fixed-length PRIMARY and SECONDARY records.

This example reveals that there is a *level of abstraction* that separates an INDEX record from its materialization as a PRIMARY and zero or more SECONDARY records. It is easy to draw data structure, field definition, and instance diagrams that occur at each level. Figure 3 shows the diagram at the upper level; Figure 4 shows the lower level. Such levels of abstraction are *not* present in the generalized UM or any of its derivatives and predecessors. Unless levels of abstraction are introduced, one is forced to model the inverted file in a single (one level of abstraction) data structure diagram. Figure 5 shows the difficulties that arise when this is tried. It is easy to identify the three internal files PRIMARY, SECONDARY, and DATA. It is also easy to identify link L which connects PRIMARY to SECONDARY. But what about link I? Since the pointers that define the parent field (i.e., inverted list) of link I are strewn over

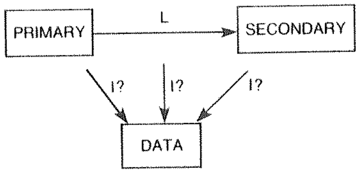


Fig. 5. The need for multiple levels of abstraction.

PRIMARY and SECONDARY records, how is one to decide the parent record of a link *I* occurrence? What is the parent file of link *I*? Three possible ways are shown in Figure 5, but it is obvious that none conveys the correct structure or relationship.

The general problem is clear. Elementary storage structures can be used to implement other elementary storage structures (e.g., list linksets are used to implement inverted list linksets). Levels of abstraction are needed to separate these structures in order to describe them in a meaningful way.

New modeling techniques, quite different from those used previously, are needed to account for the above implementation possibilities and to predict their generalizations. Central to these techniques is the idea of conceptual-to-internal mappings.

3. THE TRANSFORMATION MODEL

A primary function of a DBMS is to map conceptual files and operations to their internal counterparts. INGRES [76], for example, maps relations and relational operations onto inverted files. SYSTEM R [24] and RAPID [83] also begin with relations, but SYSTEM R maps to inverted files with record clustering and RAPID maps to transposed files.

An intuitive understanding of conceptual-to-internal mappings is gained by recognizing a mapping 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. In this way physical databases can be modeled 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 the author has studied, only ten different primitive mappings, henceforth called *elementary transformations*, have been utilized. Elementary transformations can be used singly or in combination to map files and links from one level of abstraction to a 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 files and links that the DBMS supports and (2) applying a well-defined sequence of elementary transformations to produce the internal files and 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 links). We explain each of the elementary transformations in detail shortly.

Conceptual-to-internal mappings are related to the UM in the following way. The UM relies on decomposition to identify the internal files and links of a physical database. In contrast, the TM starts with conceptual files and links that are supported by a DBMS and shows how their underlying internal files and links are derived. The TM does *not* introduce new simple file structures or linkset structures. Rather, the TM extends the UM by supplanting the intuitive process of physical database decomposition with conceptual-to-internal mappings. Thus, the primitives for describing DBMS architectures are (1) simple files, which map internal files to pages on secondary storage; (2) linksets, which specify how related records of different files are physically connected; and (3) elementary transformations, which define how abstract (or higher level) files and links are mapped to concrete (or lower level) files and links.

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. In turn, the physical database software of a DBMS can be understood as a sequence of these layers, where the software of different DBMSs are described by different sequences. 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 explain in Section 5 how the TM is being used to develop a system whose goal is to automate the development of the physical database software of DBMSs.

3.1 Elementary Transformations

Elementary transformations are rules for mapping files and links at one (higher) level of abstraction to those at the next lower (more concrete) level. Ten elementary transformations are presently recognized. They were discovered as a consequence of studying the storage architectures of SPIRES [74], DMS-1100 [73], TOTAL [20], MRS [49], IDMS [22], INGRES [76], IMS [44], ADABAS [32], INQUIRE [45], RAPID [83], ALDS [14], CREATABASE [61], and SYSTEM 2000 [16]. Models of the storage architectures for most of these systems have been completed. Table I lists the transformations that are used in each model, and a reference to the model. Preliminary models of the remaining systems—IMS, TOTAL, SPIRES, and CREATABASE—are given in [8]. Although there is ample evidence that the transformations identified in this paper are the most common, there may be other transformations which have not yet been recognized. We address the completeness issue later in Section 5.

The transformations themselves were defined to coincide with familiar physical database concepts or with their generalizations. For example, there is a transformation called segmentation which corresponds to the well-known concept of segmentation [56]. Thus, there is reason to believe that similar sets of transformations would have been identified if models of conceptual-to-internal mappings had been developed independently of our research.

Table I. Usage of Elementary Transformations in Existing Models

Elementary transformation	Database Management System								
	ADABAS	ALDS	DMS-1100	IDMS	INGRES	INQUIRE	MRS	RAPID	SYSTEM 2000
Augmentation	x	x	x	x		x	x	x	x
Encoding	x			x	x			x	
Extraction	x	x				x	x	x	x
Collection						x			x
Segmentation	x	x	x	x	x	x		x	x
Division	x		x	x		x	x		x
Actualization	x								
Layering				x				x	
Null	x			x	x	x		x	x
Horizontal partitioning									
Model reference	Appendix III	[8]	[8]	[7]	[9]	Sect. 4	[7]	[9]	Appendix IV

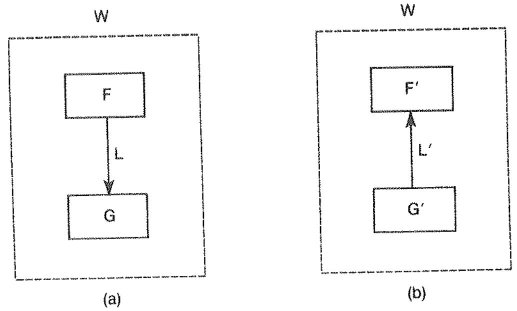


Fig. 6. Two materializations of abstract file *W*.

To illustrate and explain the effects of each transformation, we again use data structure, field definition, and instance diagrams. Besides the usual conventions, there are two additions. First, abstract objects (typically files) are indicated by dashed outlines in data structure diagrams. Figure 6a shows a data structure diagram of an abstract file *W* and its materialization as the files *F* and *G* and link *L*. Figure 6b shows another example materialization of *W* as the files *F*′ and *G*′ and link *L*′, where *L*′ has opposite directionality.

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 transformed, we rely on the orientation of record types within a dsd. The orientation of *F* and *G* in Figure 6a, for example, 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 known exception that we are aware of is full transposition, which is discussed later in the segmentation section. 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.

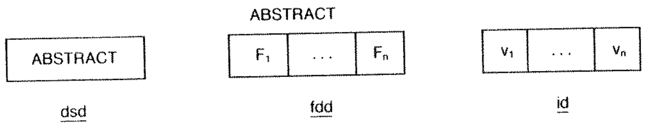


Fig. 7. An ABSTRACT record type.

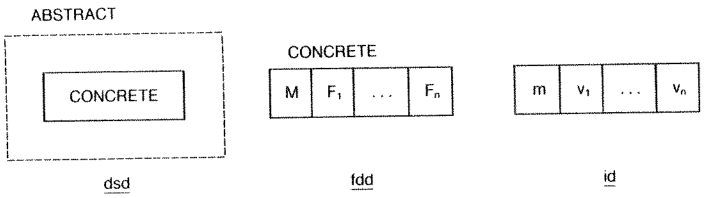


Fig. 8. Augmentation of metadata field *M*.

Note that dominance has nothing to do with links and their directionality. In Figure 6a, *F* is dominant and is the parent file of link *L*. In Figure 6b, *F*′ is dominant and is the child file of link *L*′. Thus, dominance is indicated only by being above all other files.

With the aid of these notations and the linkset structures of Appendix II serving as a basis, nine of the elementary transformations are explained and illustrated below. A tenth (horizontal partitioning) is briefly discussed in Section 5. Our illustrations of these transformations are only examples; each transformation can have many additional uses.

Augmentation (*of Metadata*). Metadata can be added to an abstract record. For example, it can be a delete flag or a record type identifier. It may be stored in a separate field or added to an existing field. A metadata field is given a name so that it may be referenced later.

Figure 7 shows a data structure, field definition, and instance diagram of a file of type ABSTRACT. An ABSTRACT record has *n* fields $F_1 \dots F_n$, with value v_i stored in field F_i. Figure 8 shows the result of augmenting metadata field *M* (with value *m*) to an ABSTRACT record. RAPID, INQUIRE, ADABAS, and SYSTEM 2000 use augmentation.

Encoding. Abstract records or selected fields thereof can be encoded for purposes of data compression, data encryption, or searching (e.g., SOUNDEX

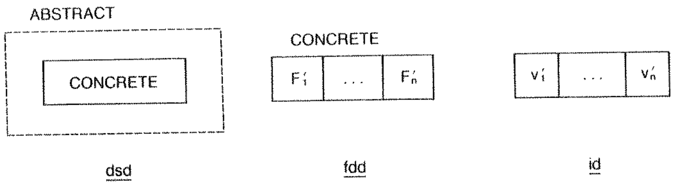


Fig. 9. Encoding of individual fields.

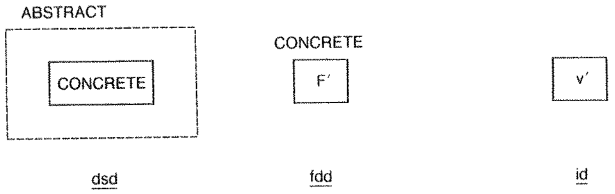


Fig. 10. Encoding of an entire record.

encoding [85]). Common data compression algorithms include the elimination of trailing blanks and leading zeros, storing numeric character strings as binary integers, digraph encoding schemes (where commonly occurring character pairs are encoded into single bytes [84]), Huffman encoding [43], and Ziv–Lempel encoding [89]. Well-known encryption algorithms are block ciphers [52] and the NBS data encryption standard [60].

Encodings are applied to individual fields or to entire records (viewed just as a string of bytes). The former allows direct access to compressed fields and compressed data values, the latter requires record expansion before specific fields can be located. Figures 9 and 10 illustrate the notation that is used to distinguish these cases on the ABSTRACT record of Figure 7. Figure 9.fdd shows unencoded field F_i mapped to encoded field F'_i, and Figure 10.fdd shows the string of fields $F_1 \ldots F_n$ mapped to a single encoded field F'. ADABAS compresses fields separately; IDMS compresses entire records.

Note that some encoding schemes, such as Huffman and Ziv–Lempel encodings, require the use of translation tables. These tables would be maintained by the system as part of the internal representation of the schema in which the ABSTRACT record type was defined. Such tables are not shown in the diagrams of Figures 9 and 10.

Extraction. Creating a secondary index on a field of an abstract record type is one of several uses of extraction. The basic idea is to extract the set of all distinct data values that appear in specified fields of abstract records.[1] Normally, one field per record type is extracted for each application of the transformation. Abstract records are mapped to concrete records and an "index" record type is created. Each extracted data value is stored in a distinct index record. Each index record is related to all concrete records that possess its data value. This relationship is realized by a link that connects the parent index file to the child concrete file. Figure 11.dsd shows the files and links that result from extracting field F_i from the ABSTRACT record of Figure 7. Note that, as a general rule, index record types are not dominant.

[1] Compound fields may also be extracted. A *compound field* is an ordered sequence of two or more elementary fields.

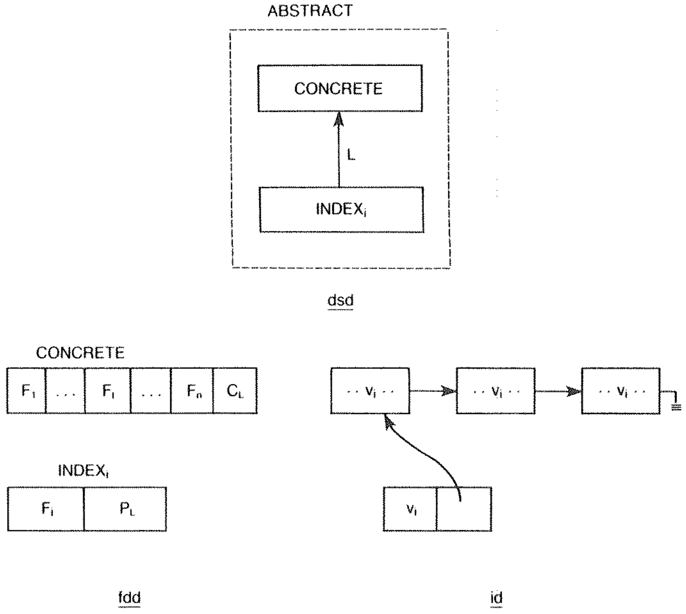


Fig. 11. Extraction of field F_i with duplication.

There are two known variations of extraction. Figures 11.fdd and 11.id illustrate extraction *with duplication* where the extracted field F_i appears in both the $INDEX_i$ and CONCRETE records. Link L in Figure 11.id is shown as a list linkset. (Other linkset implementations are possible.) ADABAS, MRS, SYSTEM 2000, and INQUIRE use extraction with duplication to create indices on data fields.

The other variation is extraction *without duplication* (i.e., the extracted field is removed from CONCRETE records). Figure 12 illustrates this transformation. Extraction without duplication is primarily used to create dictionaries rather than indices. A *dictionary* for field F_i is a lexicon of data values that defines the domain of F_i; there are no pointers or linkages that connect a data value of the dictionary to all of its occurrences in concrete records. (In contrast, an index has

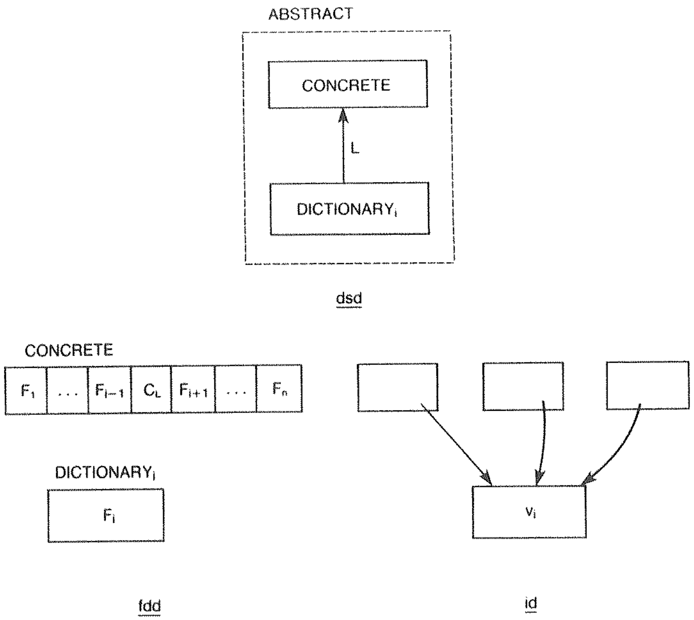


Fig. 12. Extraction of field F_i without duplication.

such linkages). Link L in Figure 12.id is implemented solely by parent pointers (i.e., pointers from child records to parent records). CREATABASE, ALDS, and RAPID use extraction without duplication to create dictionaries on data fields.

It is usually the case that DBMSs allow most, if not all, fields to have dictionaries or secondary indices. This can be modeled by repeated applications of extraction, once for each specified field. To indicate multiple extractions in a compact way, we use a special notation. In Figure 13.dsd, ( )^j is used to indicate that the INDEX$_j$ file and its link I_j can be reproduced any number of times, each time with a different value for j.

When multiple links are generated, multiple child fields (one for each link) may be introduced to the CONCRETE record. In Figure 13.fdd, the presence of multiple child fields in the CONCRETE record type is shown by $(C_{I_j})^j$, where, again, ( )^j is the repeat notation. The repeat notation is also used in Figure 13.fdd to indicate the generation of multiple INDEX$_j$ record types. There is, of course, an implicit coordination between data structure and field definition diagrams which use the repeat notation (i.e., the values of j used in the dsd are identical to those used in the fdd).

Some additional points need to be stressed. First, whenever a link is introduced by a transformation, it may be realized in principle by any linkset structure—list, pointer array, sequential, or relational (see Appendix II). As mentioned previously, ADABAS, MRS, SYSTEM 2000, and INQUIRE use extraction with duplication. The link that is produced is realized by pointer arrays (inverted lists) in ADABAS, MRS, and SYSTEM 2000; it is realized by lists in INQUIRE.

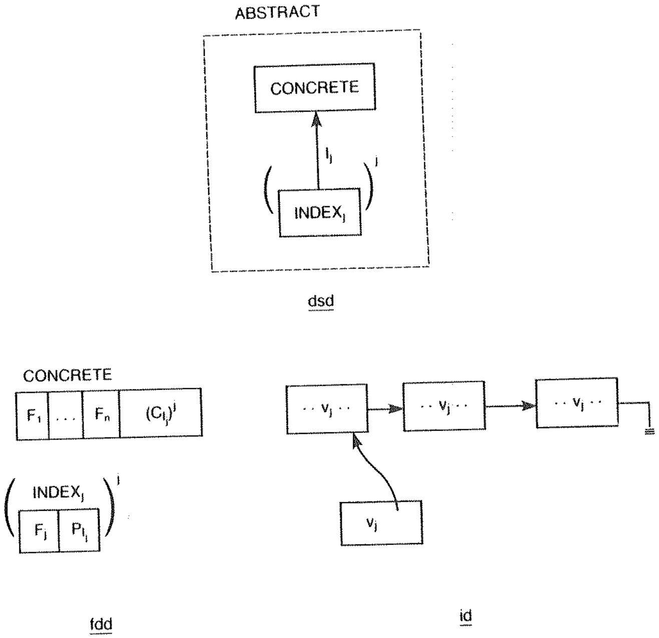

ABSTRACT
CONCRETE
I_j
INDEX$_j$
j
dsd
CONCRETE
F_1 ... F_n $(C_{I_j})^j$
INDEX$_j$
F_j P_{I_j}
j
$\cdot\cdot v_j \cdot\cdot$
$\cdot\cdot v_j \cdot\cdot$
$\cdot\cdot v_j \cdot\cdot$
v_j
fdd
id

Fig. 13. The repeat notation.

Second, we use the term "extraction" rather than "indexing" since this transformation is used for purposes other than creating indices (e.g., dictionaries and phantom files [85]).

Third, the extraction transformation can be applied to *derived fields* (i.e., fields that are not actually present in a record, but whose value(s) can be computed by applying an "extraction" function to the record itself, see [78], [77], [82], [63]). A simple example of a derived field would be the calculation of monthly salaries, given yearly salaries. As a more complicated example, a text field could be "indexed" by applying a function to the field which returns the set of key words that it contains. An index record would then be defined for each distinct key word. As another example, a record could describe an object located on a circuit diagram. To support window retrieval (i.e., retrieval of all components of a diagram that fall within a specified geometric region), a circuit is partitioned into subcells. An "extraction" function applied to a record would produce the set of subcells in which its corresponding object lies. These subcell references could

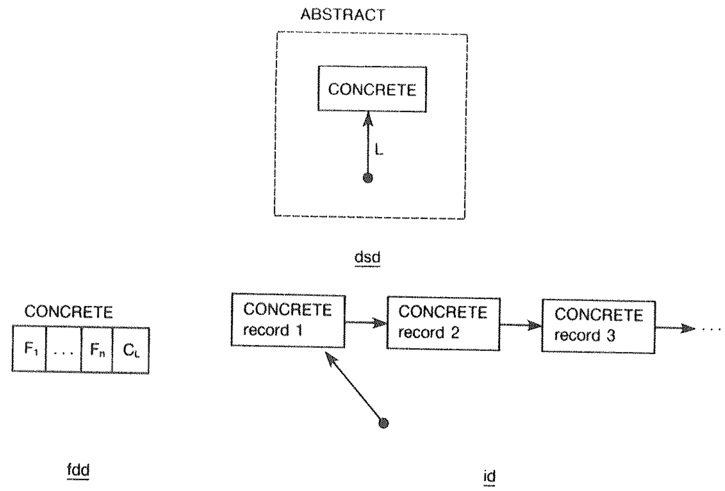


Fig. 14. Collection.

then be used to optimize window retrieval [36]. Notions of extraction with and without duplication are extended accordingly.

Collection. The DBTG concept of a singular set, which links together all records of a given type, is an example of the collection transformation. The basic idea is to collect all instances of one or more abstract record types together onto a single link occurrence. Figure 14 shows the result of collecting the ABSTRACT record type of Figure 7. Note that the parent record of the lone L occurrence is maintained as part of the internal representation of the schema in which the ABSTRACT record type was defined. It is indicated by "•" in Figure 14.dsd.

In all applications of collection known to the author, link L has been realized by a list linkset (see Fig. 14.id), although other linksets conceivably might be used.[2] INQUIRE and SYSTEM 2000 use this transformation.[3]

Segmentation. Abstract records can be partitioned along one or more field boundaries to produce two or more subrecords. One subrecord is distinguished as the primary record, the rest are secondary records. A link connects the primary file to each secondary file. Primary records are differentiated from secondary records as they are materialized and processed differently. Usually it is the case that the most active fields (i.e., the ones that are retrieved and updated most frequently) are placed in the primary record and the remaining are in the secondary [56].

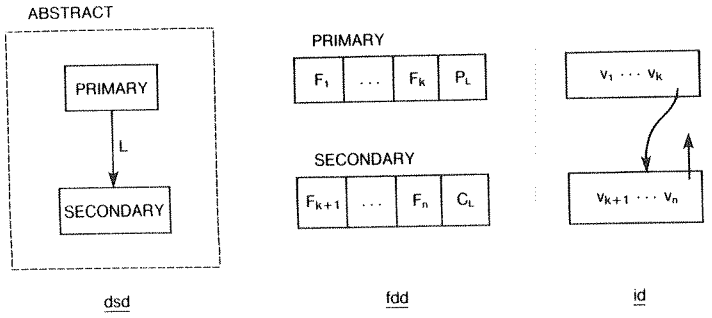


Fig. 15. Segmentation without duplication.

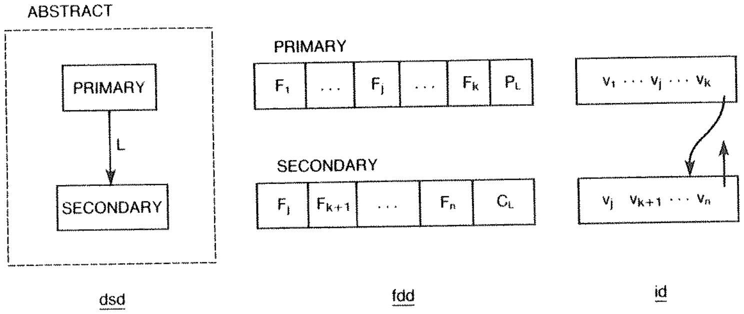


Fig. 16. Segmentation with duplication.

Segmentation can occur with or without duplication of data fields. Figure 15 shows the segmentation of fields $F_1 \ldots F_k$ from $F_{k+1} \ldots F_n$ of the ABSTRACT record of Figure 7. No fields are duplicated, and L is shown as a *singular pointer* (i.e., a pointer array with precisely one child pointer) and a parent pointer. The same segmentation occurs in Figure 16, except that field F_j is duplicated. RAPID and IMS use segmentation with duplication; ADABAS and INGRES use segmentation without duplication.

Two forms of segmentation without duplication are so well-known or occur so frequently that they have been given special names. One is *full transposition*, which segments each field into separate subfiles. That is, if there are n fields in an abstract record type, then a full transposition produces n concrete record types, each containing precisely one field (see Figure 17). Because all fields are

[2] A rather odd implementation of link L would be as a sequential linkset. The system record • would be immediately followed by all CONCRETE record occurrences. Note that the resulting linkset occurrence does *not* define a simple file structure. The • record could be stored in a hash-based, indexed-sequential, etc., structure and its train of CONCRETE records would then follow. Linksets connect parent records to their child records; simple file structures map records into blocks.

[3] A generalization of collection was proposed in DIAM [67]. It would collect all records (of possibly several types) that satisfy a predicate onto a single link occurrence.

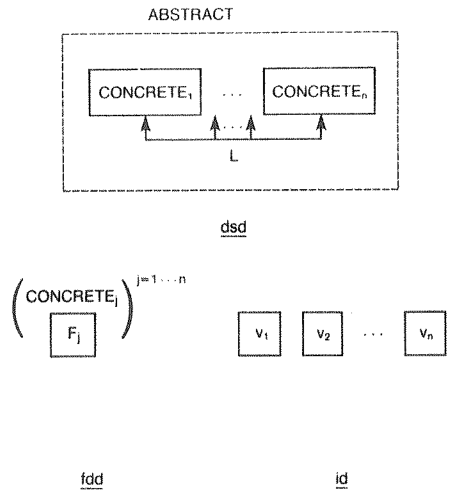


Fig. 17. Full transposition.

treated identically, the resulting concrete types are not distinguished as being either "primary" or "secondary". Thus, all may be considered as dominant. (That is, a pointer, to an abstract record can serve as a pointer to any of its transposed subrecords). Note that link L in Figure 17.dsd, which interconnects the n concrete types, is drawn as a bidirectional link which does not force "parent" and "child" distinctions. Link L is implemented as by a transposed linkset, which is described in Appendix II. Further information on transposed files can be found in [4, 40, 56]. RAPID and ALDS use full transposition.

Full transposition represents one extreme form of segmentation. Another is the second well-known form, called *indirection*, where all fields are removed to a secondary record and only a pointer remains in the primary. An INDIRECTION record and a CONCRETE record connected by link L is a result (see Figure 18). The INDIRECTION record contains only the field P_L; the CONCRETE record contains all the fields of the abstract record and (optionally) field C_L. Figure 18.id shows L as a singular child pointer and a parent pointer, although there are other variations. DMS-1100 uses only singular pointers, and ADABAS uses a cellular singular pointer (a pointer that references the block in which the CONCRETE record is stored) and a parent pointer.

As mentioned earlier, it is common for pointers to reference abstract records. The goal of the indirection transformation is to be able to alter the storage location of a CONCRETE record without having to update pointers to its corresponding abstract record. This is accomplished by fixing the storage location of the INDIRECTION record and updating the P_L pointer each time its CONCRETE record moves.

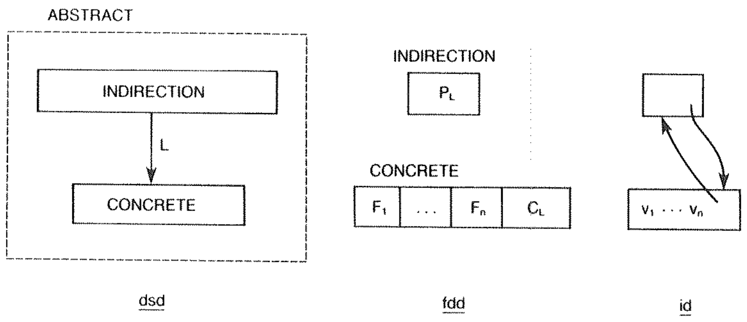


Fig. 18. Indirection.

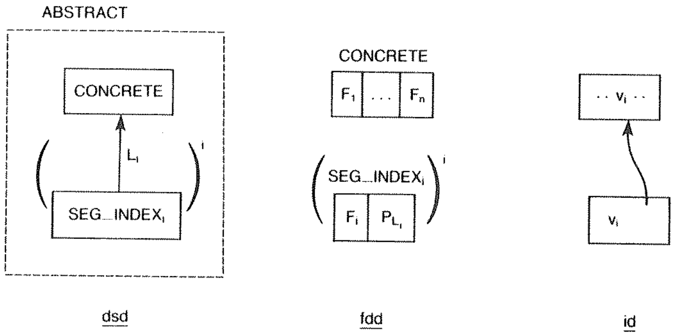


Fig. 19. Segmented secondary indices.

Yet another common use of segmentation is to create files that function as secondary indices. If a "segmented" secondary index for field F_i is to be created, F_i is segmented with duplication from ABSTRACT records to produce a SEG_INDEX$_i$ file connected to a CONCRETE file via link L_i, as shown in Figure 19. Link L_i is usually implemented by a singular pointer (Figure 19.id). (Note that the primary SEG_INDEX$_i$ file is not dominant; in all previous examples of segmentation, primary files were dominant.)

By this construction, it follows that the number of SEG_INDEX$_i$ records always equals the number of CONCRETE records. This means that if some value v_i occurred, say, twenty times, there would be twenty SEG_INDEX$_i$ records that contained value v_i. Note that this form of indexing is different than the secondary indices produced by extraction. (In extraction, there would be only one index record that contained value v_i, no matter how many times v_i occurred in the CONCRETE file). Also, the algorithms that support "segmented" secondary indices would be different than those that support "extracted" indices. INGRES and RAPID use segmented secondary indices.

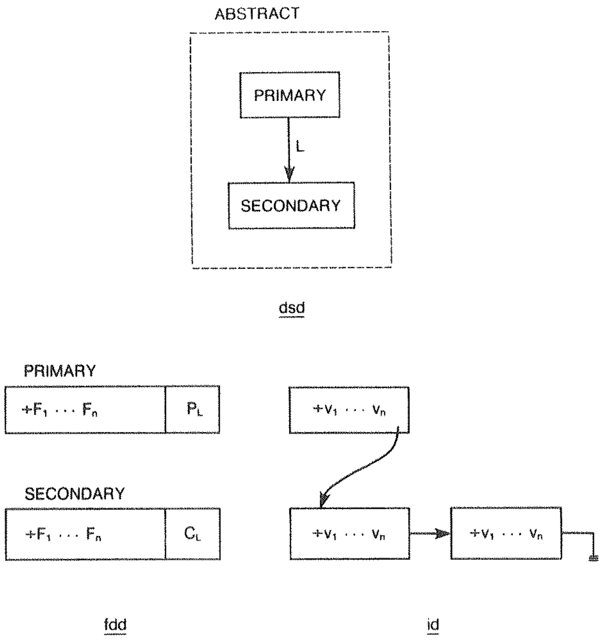


Fig. 20. Division without duplication.

It is interesting to note that segmented secondary indices and extracted secondary indices are equifrequent in DBMS implementations. Although it is believed that segmented secondary indices are easier to implement, it is not known which of the two methods is more efficient.

In connection with segmented secondary indices, segmentation can also be applied to derived fields, just as extraction can be applied to derived fields. A list of possible applications was given earlier in the section on extraction.

Division. Division is the partitioning of an abstract record or of selected fields into two or more fragments. The first fragment is the primary (and dominant) fragment, and the remaining are secondary fragments. Unlike segmentation, partitioning is done without respect to field boundaries. A record or field is usually divided into fixed-length fragments (e.g., the first hundred bytes define fragment 1, the next hundred bytes fragment 2, and so on). Division is otherwise identical to segmentation.

Division may occur with or without duplication of fields. Figure 20 shows the result of applying division without duplication to the ABSTRACT record of Figure 7. Figure 21 shows the division of the same ABSTRACT record with the duplication of field F_1 in each fragment. (Note that $\div F_2 \ldots F_n$ denotes a fragment of the string of fields $F_2 \ldots F_n$. Each fragment does not overlap with other fragments. Taken together these fragments can be concatenated to form the original string.)

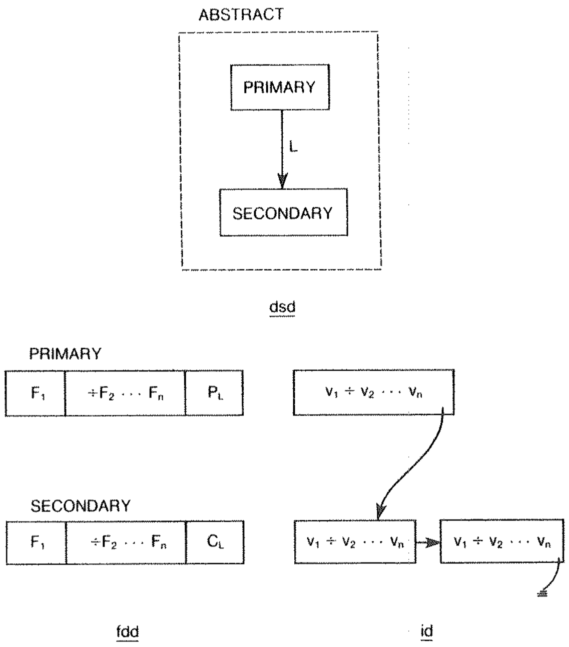


Fig. 21. Division with duplication.

INQUIRE and SYSTEM 2000 use list linksets to realize link L which connects the PRIMARY file to the SECONDARY file (see Fig. 20.id). SYSTEM R uses pointer arrays to realize L in the implementation of long fields [38] and map arrays for complex objects [53]. ADABAS uses relational linksets (i.e., records are related by sharing common keys).

We have already seen an example of division: Figure 4 shows the division of an INDEX record into fragments connected by a list linkset. Another common use of division, this time combined with relational linksets, arises when conceptual records of a database are much larger than what can be handled by the DBMS itself. Figure 22 shows how a CONCEPTUAL record with primary key k is mapped by division with duplication to four concrete records with key k duplicated in each fragment. The primary key of the jth fragment is the ordered pair (k, j). The first fragment is a PRIMARY record (with key $(k, 1)$) and the remaining fragments are SECONDARY records. The fragment numbers (which, incidentally, are stored in the parent and child fields of linkset L) specify the

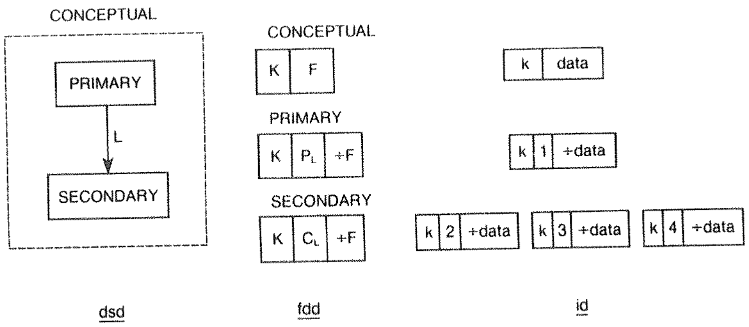


Fig. 22. Division and relational linksets.

ordering of the fragments.[4] Thus, to retrieve a "long" CONCEPTUAL record, one retrieves its corresponding PRIMARY and SECONDARY records and concatenates the fragments. Figure 22 is a good example of how an implementation "trick" can be expressed in terms of elementary transformations and linkset implementations.

Actualization. Actualization maps an abstract link to one or more concrete links and zero or more concrete files. Perhaps the most common example of actualization is the materialization of *M:N* links in DBTG databases. Consider conceptual files *F* and *G* which are related by an *M:N* link *L* (see Figure 23a). In a DBTG DBMS, link *L* would be expressed by two 1:*N* links (i.e., sets) *F-FG* and *G-FG* and a file *FG* (see Figure 23b). Links *F-FG* and *G-FG* and file *FG* can be implemented in a variety of ways (see [22, 73]). In this example, note that the mapping of link *L* is *not* accomplished by the DBMS, but rather by the database administrator when he defines the DBTG schema. Thus database users recognize Figure 23b as the DBTG implementation of Figure 23a. In principle, however, a nonDBTG network DBMS could be written which would handle this mapping automatically.

Actualization can be with or without field duplication. Normally it is without. With duplication, selected fields of a parent record type can be copied into its child record types and vice versa. Depending on the cardinality of the parent-child relationship (i.e., 1:1, 1:*N*, and *M:N*) and the cardinality of the fields themselves (i.e., scalar or repeating), the fields that are copied may contain single data values or they may have a variable number of values.[5] ADABAS uses actualization without duplication.

[4] Following the linkset terminology of Appendix II, *L* is a relational linkset with child records maintained in sort-key order.

[5] It is worth noting that the idea of actualization was considered some time ago in a rather different context. Mitoma [58] and Berelian and Irani [12] addressed a DBTG database design problem. Their approach was to start with binary data model of the database. By iteratively applying what we call actualization transformations, a DBTG schema was produced.

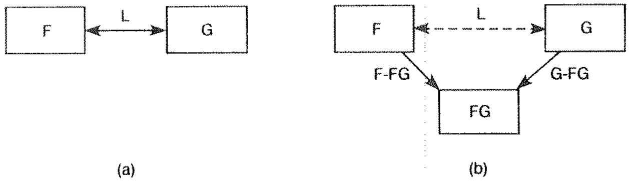


Fig. 23. Actualization of conceptual links.

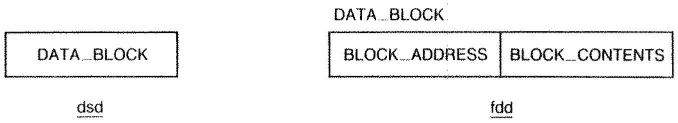


Fig. 24. Layering.

Layering. Simple file structures map internal files to blocks (pages). In some DBMS storage architectures, two types of blocks are recognized: logical and physical. It is usually the case that several logical blocks can fit into one physical block. To understand how logical blocks are mapped to physical blocks, it is necessary to model storage architectures in layers, where each layer has well-defined notions of internal records, file structures, and blocks. The upper layer has logical blocks, and the lower has physical blocks. A block on the upper layer is treated as an abstract record on the lower layer (Figure 24); the storage address of the block is the abstract record's primary key. Thus a block fetch on the upper layer is mapped to a record read on the lower; a block update on the upper layer is mapped to a record update on the lower. Elementary transformations are used to map these abstract records to internal files, and simple file structures map these internal files to physical blocks. It is in this way that "logical blocks" are mapped to "physical blocks." IMS and RAPID rely on layering to map virtual address spaces to a physical address space.

The most common use of layering is found in the file systems of operating systems. UNIX, for example, provides the abstract view of a secondary storage file as a sequence of bytes. In reality, UNIX treats contiguous sequences of 512 bytes as fixed-length records and stores them on disk in usually nonsequential locations using the standard UNIX file structure [64]. DBMSs that rely on UNIX files, such as INGRES and MRS, define contiguous sequences of 2048 or 512 bytes as (logical) blocks and use them to build unordered, B+ tree, and indexed-sequential file structures. Thus a (logical) block fetch at an upper layer (i.e., DBMS software) becomes one or more record reads at a lower layer (i.e., UNIX software).

Null. Abstract records are normally subjected to one or more transformations before their materialization has been specified. Occasionally the application of these transformations will occur only under certain well-defined conditions. For

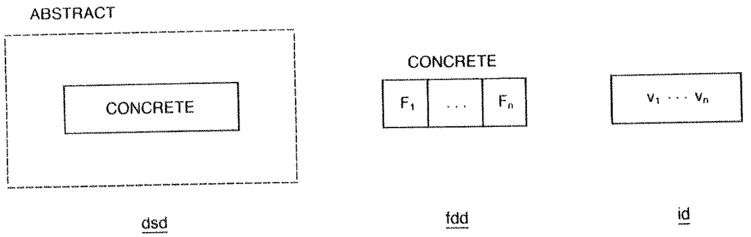


Fig. 25. Null.

example, a flag can be specified in the schema to indicate whether records of a particular type are to be compressed; the setting of this flag defines the condition on which an encoding transformation is to be applied. If conditions are not met, the abstract record is mapped directly to a concrete record without alteration. The null transformation is used to model these situations. Figure 25 shows the result of applying null to the ABSTRACT record of Figure 7. Models of the storage architectures of SYSTEM 2000 and INQUIRE utilize this transformation.

It is believed that these nine transformations are sufficient to model the storage architectures of most commercial and specialized database management systems. Since only a relatively small number of DBMSs have been examined so far, it is possible that other transformations may exist or that existing transformations can be generalized. Thus, our model should be considered preliminary.

In the following section we outline a general procedure for modeling the storage architecture of a DBMS using these transformations.

3.2 A Procedure for Modeling DBMS Storage Architectures

Most DBMSs support a logical or conceptual data model that is record-oriented. DBTG network-based systems, such as IDMS and DMS-1100, hierarchical systems, such as SYSTEM 2000 and IMS, and even relational systems, such as INGRES and SYSTEM R, have record-based models. Future DBMSs are likely to support semantic data models that are object-oriented, such as DAPLEX [70] or the Entity-Relationship model [19], in order to capture and utilize the semantics of database objects more fully [11].

The first step in modeling the storage architecture of a DBMS is to begin with a generic data structure diagram that captures the different kinds of links that the DBMS permits among conceptual files. In this paper we are not concerned with the mapping of semantic (object-oriented) data models to a record-oriented representation. Again, as almost all conventional DBMSs are record-oriented, starting with a data structure diagram is not a restrictive requirement. However, we note that the mapping of semantic data models to record-based models will eventually become an important step in modeling the storage architecture of future database systems.

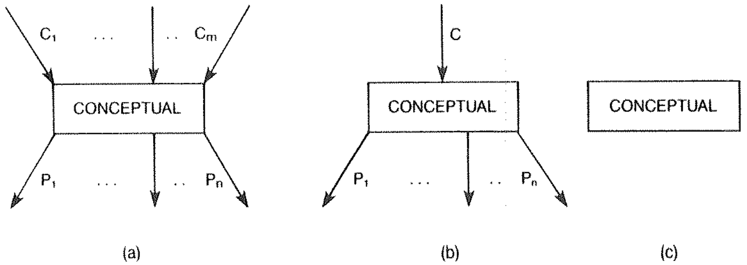


Fig. 26. Generic conceptual data structure diagrams.

Figure 26 shows three generic data structure diagrams that reflect the network, hierarchical, and single-file data models. Variations of these diagrams may be used to capture features that are peculiar to specific DBMSs. In a network DBMS, a CONCEPTUAL file can be a child file for links $C_1 \ldots C_m$ and the parent file for links $P_1 \ldots P_n$, for $m \geq 0$ and $n \geq 0$ (Figure 26a). Note that instances of such files may have different values for m and n. The generic dsd for a hierarchical DBMS is shown in Figure 26b. Instances of files in the hierarchy are the root ($m = 0$, $n \geq 1$), the leaves ($m = 1$, $n = 0$), and the intermediates ($m = 1$, $n \geq 1$). Some file management systems, such as ALDS, do not explicitly support links between conceptual files. In these cases, the generic dsd would be a single conceptual file (Figure 26c). Relational DBMSs that realize conceptual links by joins may also begin with Figure 26c.

A characteristic of conceptual-to-internal mappings is the generation of many files that are neither conceptual nor internal. In order to reference them, they will need to be given names. As a convention, we preface their names by "ABSTRACT_" so that they can be distinguished from conceptual and internal files.

The second step in modeling DBMS storage architectures is to specify the implementation of the conceptual links, perhaps using *actualization*. This step introduces parent and child fields into the record types that are related by linksets. To distinguish CONCEPTUAL records from those that contain parent and child fields, we refer to the records of the latter type as ABSTRACT_CONCEPTUAL records, using the convention of the prefix "ABSTRACT_" mentioned above.

At this point a single CONCEPTUAL or ABSTRACT_CONCEPTUAL file has been identified. The materialization of this file proceeds in well-defined steps, where one or more elementary transformations may constitute a single step. A step is usually identified with all transformations that are applied to a single file. The sequence of transformations that comprise a derivation follows an intuitively evident course in which abstract files are made progressively more concrete. This progression can be seen in any of the derivations presented in this paper. The process of applying elementary transformations terminates when the record types of internal files (i.e., the record types of the records that are stored in simple files) have been derived. The result at this stage in the architecture modeling is a set of internal files and internal links.

The final step is to assign each internal file to a simple file structure and each internal link to a linkset structure. It is at this step where blocking factors, primary keys, overflow methods, file placement, and so on are given.

It is worth noting that simple file structures often augment internal records with delete bytes and pointers, and may introduce list structures (such as overflow chains) that look quite similar to linkset implementations. Thus the question arises when to stop applying elementary transformations in modeling a storage architecture. The solution lies in the definition of the interface to simple files; all augmented fields, pointers, and so on that are not added below this interface must be handled by elementary transformations.

Unfortunately, there is much confusion in actual DBMS software in identifying such an interface. Many DBMSs were not developed in a modular fashion; "higher level" routines directly manipulate "lower level" details, thereby obscuring the simplicity of a layered implementation. Other DBMSs have clearly identifiable software layers, but their boundaries differ substantially from those required by the TM and UM.

We have implemented a file management system, called JUPITER, that is based on the simple file submodel of the UM [31]. JUPITER is consistent with the concepts of internal files and simple files used in this paper. With the JUPITER interface, it is obvious whether functions should be supported by simple files or by elementary transformations. The storage architecture models that we present in this paper are consistent with this interface.

One final note concerns the representation of conceptual records. We view a conceptual record simply as a sequence of values. In reality, it is a string of bytes which defines the DBMS's input/output representation of these values. This might involve the use of ASCII or EBCDIC codes, or the use of special data structures (e.g., pointers or count bytes) to separate the contents of repeating or variable length fields (see [57]). The actual encoding that a DBMS uses to input and output its records is irrelevant to understanding the DBMS's storage architecture. For this reason we ignore such encodings.

In the following section, and in the appendices, we apply this procedure to model the storage architectures of INQUIRE, ADABAS, and SYSTEM 2000. We have chosen INQUIRE as our main example, for it is representative of the complexity of most DBMS architectures and is a good illustration of how implementation "tricks" can be expressed as conceptual-to-internal mappings. The storage architectures of ADABAS and SYSTEM 2000 are presented in Appendices III and IV. References to other architectures are given in the Introduction.

In each of the examples, a considerable amount of detail is progressively revealed. Although many details may seem unimportant and some of the implementation methods are clearly nonoptimal, it is precisely these details and methods that one must understand in order to comprehend the implementation of these DBMSs. The purpose of these examples is to demonstrate that the TM is powerful enough to model practical systems.

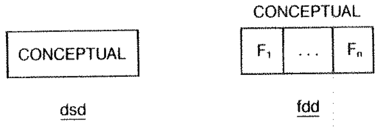


Fig. 27. Generic CONCEPTUAL record type of INQUIRE.

4. THE STORAGE ARCHITECTURE OF INQUIRE

INQUIRE is a product of Infodata Systems Inc. It is presently used in more than 300 installations in North America and Europe. INQUIRE creates a distinct physical database for each conceptual file that is defined. Interconnections between different conceptual files are implicit; they are realized by join operations rather than by physical structures. The underlying storage architecture of INQUIRE, therefore, can be understood by examining how records of a single conceptual file are stored.

The generic CONCEPTUAL record type supported by INQUIRE is shown in Figure 27. It consists of n fields, $F_1 \ldots F_n$, which may be elementary or compound. The value of n is user-definable. An elementary or compound field may be scalar or repeating. A *scalar* field always contains a single data value (possibly null). A *repeating* field contains zero or more data values. Data values can have fixed or variable lengths. Thus CONCEPTUAL records are typically variable-length.

CONCEPTUAL record types are the record types that are defined in INQUIRE schemas; CONCEPTUAL records are the records that are visible to INQUIRE users.

The internal files and links of INQUIRE are derived in the following way. First, INQUIRE *augments* a delete flag DF to every CONCEPTUAL record. This flag is used to mark CONCEPTUAL records that have been deleted. Next, INQUIRE allows scalar and repeating fields to be indexed. Field F_j is indexed by *extraction*. This produces the ABSTRACT_INDEX$_j$ and ABSTRACT_CONCEPTUAL files connected by link I_j (Figure 28). Thus, for each distinct data value that appears in field F_j in one or more ABSTRACT_CONCEPTUAL records, there will be a distinct ABSTRACT_INDEX$_j$ record that contains this value.

INQUIRE creates indices for scalar and repeating fields in the same way. Figure 28.id illustrates the indexing of a repeating field. Three ABSTRACT_CONCEPTUAL records and two ABSTRACT_INDEX$_j$ records are shown. Although each ABSTRACT_CONCEPTUAL record contains many data fields, only the contents of repeating field F_j are shown; one record contains a value v_1, another contains v_1 and v_2, and a third contains v_2. The ABSTRACT_INDEX$_j$ records shown are those for values v_1 and v_2. Note that the ABSTRACT_CONCEPTUAL record whose F_j field contains v_1 and v_2 has both ABSTRACT_INDEX$_j$ records as its parents. Thus link I_j is $M{:}N$.[6]

[6] So that there is no ambiguity about the distinction between 1:N and $M{:}N$ links, it is well known that CODASYL sets are 1:N. If $M{:}N$ sets were supported, a member record could participate in multiple occurrences of the *same* set at the same time. Link I_j is equivalent to an $M{:}N$ set.

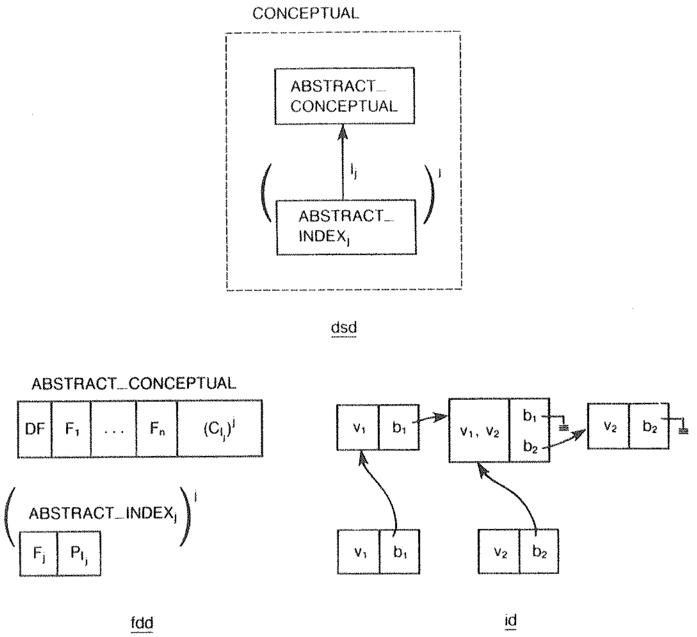


Fig. 28. Augmentation and extraction of CONCEPTUAL fields.

The linkset that implements I_j is an M:N multilist in which child records are chained in descending physical address order. N:M multilists are implemented by assigning a distinct fixed-length binary value, called a *binkey*, to each list occurrence. A binkey is paired with each pointer of its list so that pointers of one list can be distinguished from those of another. In Figure 28.id, the binkey of the multilist for data value v_1 is b_1 and the binkey for v_2 is b_2.

It is important to note that the child field C_{I_j} of link I_j is repeating. The number of elements in a C_{I_j} field equals the number of data values that the record has in field F_j. The repeating element is a binkey-pointer pair. Thus the first ABSTRACT_CONCEPTUAL record of Figure 28.id has a C_{I_j} field with one binkey-pointer pair (the binkey is b_1), the second has two (both b_1 and b_2 are present), and the third has one (its binkey is b_2).[7]

Any number of fields can be indexed. This is shown in Figures 28.dsd and 28.fdd by the use of the repeat notation. As defined in Section 3.1, it means that if m fields are extracted, there will be m ABSTRACT_INDEX files, each connected to ABSTRACT_CONCEPTUAL by precisely one link. A total of m child fields would appear in the ABSTRACT_CONCEPTUAL type, one for each link that is generated.

INQUIRE requires indexed fields to be designated as being either *prefix* or *simple*. The distinction is evident to a user at the query language level where an equality predicate on a prefix field must be expressed as "field name = value," whereas on a simple field it is merely "value." (Apparently, the distinction was made in order to allow queries on frequently referenced attributes to be expressed more compactly.)

As an illustration, consider the retrieval of all records of a CONCEPTUAL file that have the data value "TOP SECRET" in the SECURITY field. If SECURITY is prefix, the INQUIRE operation "FIND SECURITY=TOP SECRET" would accomplish the retrieval. If SECURITY is simple, "FIND TOP SECRET" would be the operation.

The distinction between prefix and simple fields is also seen in the implementation of INQUIRE. The ABSTRACT_INDEX$_j$ records for field F_j are made concrete by *augmenting* the characteristic string "F_j =" (i.e., the field name followed by an equal sign) to each F_j data value. This is done to prefix fields only (Figure 29). No augmentation (i.e., *null*) is performed on simple fields. Figure 29.id shows the results of these transformations on the ABSTRACT_INDEX$_j$ record of Figure 28.id containing value v_1.

Again, consider the SECURITY field example. Suppose two possible values of SECURITY are "TOP SECRET" and "CONFIDENTIAL." If SECURITY is prefix, the data value strings "SECURITY=TOP SECRET" and "SECURITY=CONFIDENTIAL" would be stored in distinct PREFIX_INDEX records. If SECURITY is simple, the strings "TOP SECRET" and "CONFIDENTIAL" would be stored in separate SIMPLE_INDEX records.

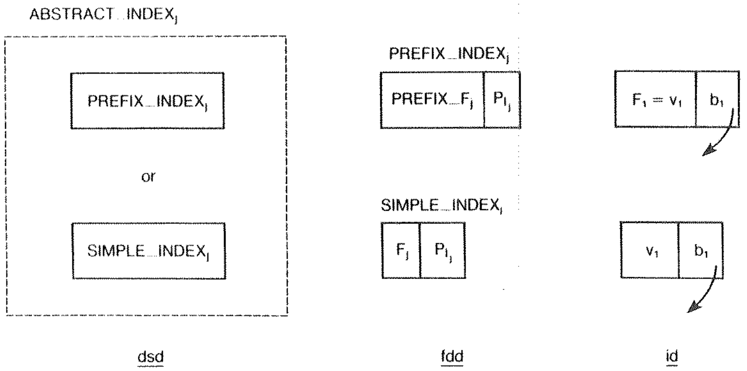


Fig. 29. Augmentation of null transformation of ABSTRACT_INDEX records.

[7] There is also a subfield in each P_{I_j} field which contains a count of the number of ABSTRACT_CONCEPTUAL records on a list. This subfield is not shown in any of our figures, but it is used in processing queries using multilists.

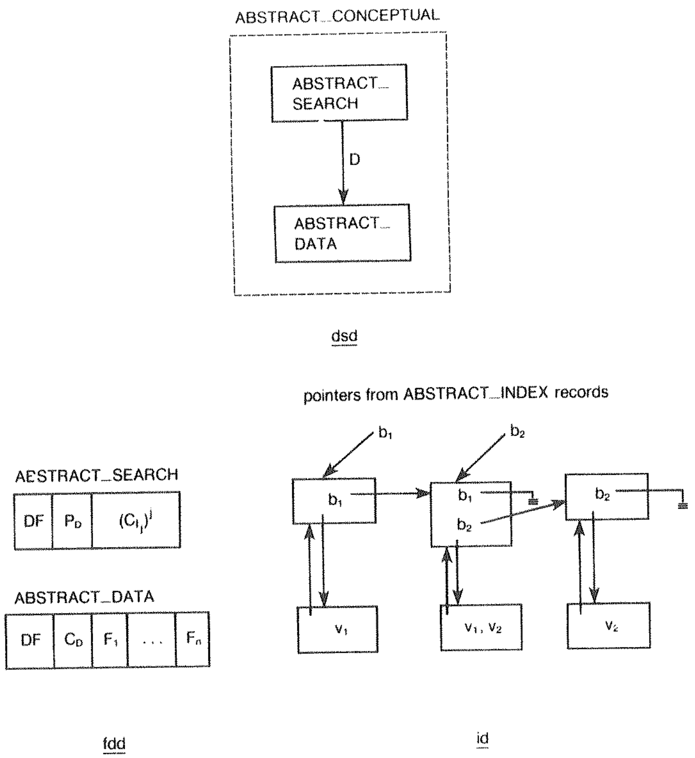


Fig. 30. Segmentation of ABSTRACT_CONCEPTUAL records.

SIMPLE_INDEX and PREFIX_INDEX are internal files. INQUIRE forces SIMPLE_INDEX and PREFIX_INDEX records to share an identical format and fixed length. This is done so that all index records can be stored in a single file structure rather than having a separate file structure for each indexed field (as is done in SYSTEM 2000, IMS, and INGRES, among others).

An ABSTRACT_CONCEPTUAL record of Figure 28 is materialized by *segmenting* all child fields $(C_{I_j})^j$ from data fields $F_1 \ldots F_n$ (see Figure 30). The delete flag DF is duplicated in both segments. This segmentation produces the ABSTRACT_SEARCH and ABSTRACT_DATA files. Link D, which connects ABSTRACT_SEARCH to ABSTRACT_DATA, is realized by a singular child pointer and a parent pointer. Figure 30.id shows the result of this segmentation on the ABSTRACT_CONCEPTUAL records of Figure 28.id.

ABSTRACT_SEARCH records are variable-length because each C_{I_j} field may contain a variable number of (binkey, pointer) pairs, one pair for each distinct value in an indexed repeating field. Rather than storing these records as is, INQUIRE *divides* an ABSTRACT_SEARCH record into fixed-length fragments. The primary fragment, which contains the P_D field of ABSTRACT_SEARCH, is a SEARCH record; all secondary fragments are SEARCH_OVERFLOW records (Figure 31). Note that the delete flag DF is duplicated in primary and secondary fragments. The SEARCH and SEARCH_OVERFLOW files are connected by link S, which is realized by a 1:N list with parent pointers. Records are maintained in order of ascending physical addresses. Figure 31.id shows an ABSTRACT_SEARCH record divided into four fragments: one SEARCH and three SEARCH_OVERFLOW. SEARCH and SEARCH_OVERFLOW are internal files.

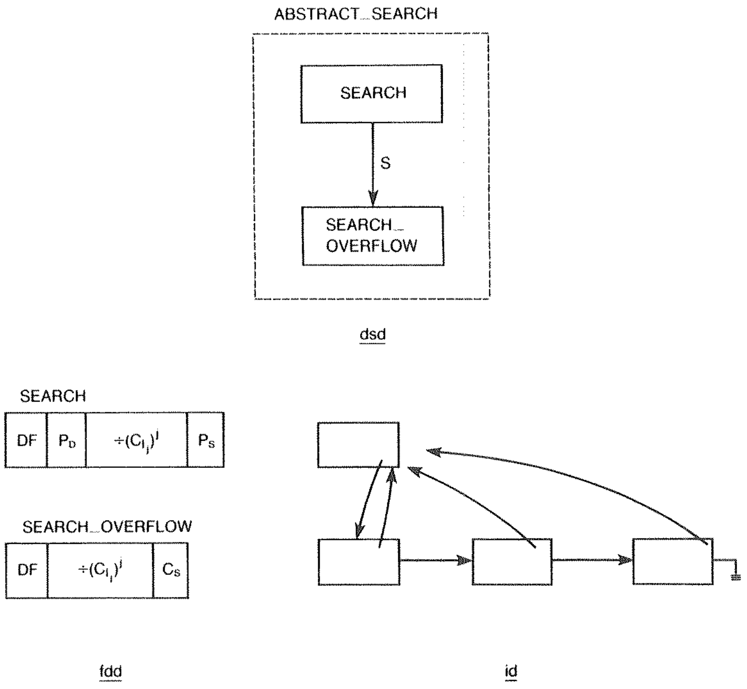


Fig. 31. Division of ABSTRACT_SEARCH records.

The ABSTRACT_DATA file of Figure 30 is materialized in two steps (see Figure 32). First, instances of ABSTRACT_DATA are usually variable-length, as some fields are repeating. INQUIRE *divides* an ABSTRACT_DATA record

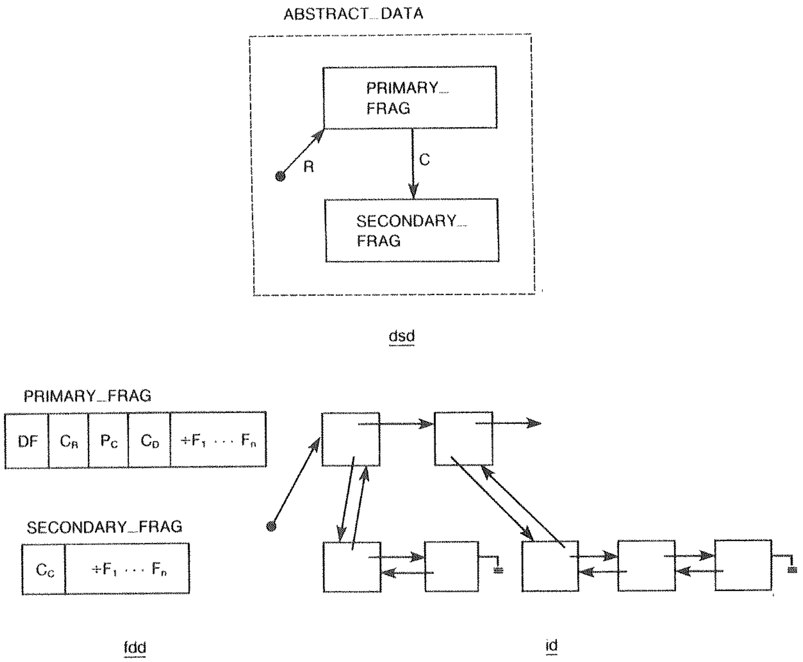
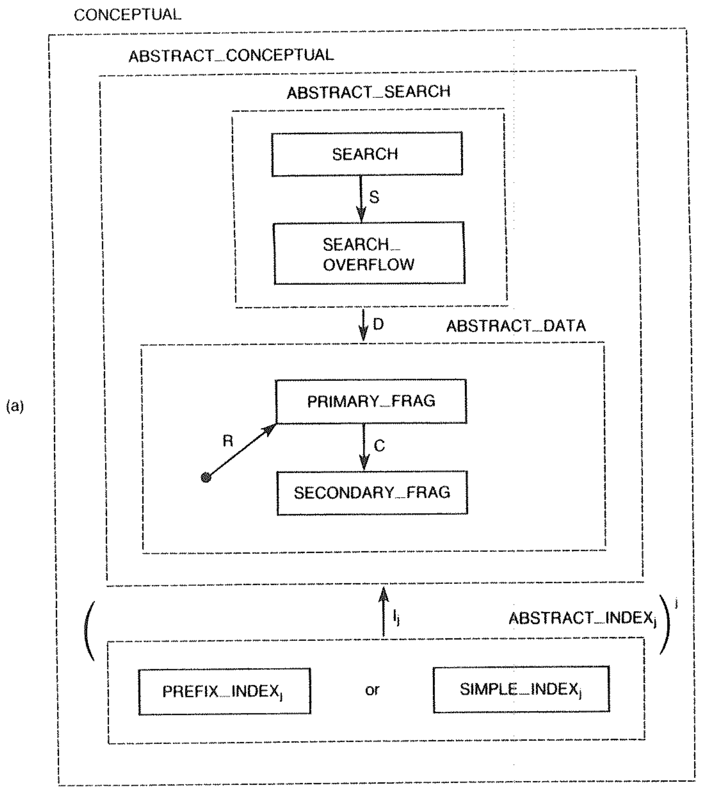


Fig. 32. Division of ABSTRACT_DATA records.

into a primary fragment (PRIMARY_FRAG) and zero or more secondary fragments (SECONDARY_FRAG) connected by link *C*.[8] *C* is realized by a 1:*N* doubly linked list with SECONDARY_FRAG records arranged in ascending physical address order. Second, all instances of PRIMARY_FRAG are *collected* onto a single list. Link *R*, which realizes the collection, is implemented as a 1:*N* list. Records are linked in reverse chronological order. Figure 32.id shows two ABSTRACT_DATA records; one is in three fragments (one primary, two secondary), the other is in four. PRIMARY_FRAG and SECONDARY_FRAG are internal files.

All occurrences of PRIMARY_FRAG and SECONDARY_FRAG are stored in a single file structure. A function of link *R* is to distinguish instances of these record types; another function is to help retrieve all CONCEPTUAL

[8] Primary and secondary fragments are variable-length. The length of a primary fragment is fixed at the time of record insertion; it equals the length of the ABSTRACT_DATA record (as it appeared initially to INQUIRE) plus some extra space. The amount of extra space can be declared as a constant or a function of the record size. As data values are added to repeating fields of an ABSTRACT_DATA record, its length may expand beyond the size of its primary fragment. It is at this point when division takes place.



Abstract File	Elementary Transformations
CONCEPTUAL	Augmentation and extraction
ABSTRACT_INDEX$_j$ (for all j)	Augmentation or null
ABSTRACT_CONCEPTUAL	Segmentation
ABSTRACT_SEARCH	Division
ABSTRACT_DATA	Division and collection

(b)

Internal File	Simple File
PREFIX_INDEX$_j$ (for all j)	INDEX_SF
SIMPLE_INDEX$_j$ (for all j)	INDEX_SF
SEARCH	SEARCH_SF
SEARCH_OVERFLOW	SEARCH_OVRFLW_SF
PRIMARY_FRAG	DATA_SF
SECONDARY_FRAG	DATA_SF

(c)

Simple File	Implementation
INDEX_SF	B+ tree or Indexed-sequential
SEARCH_SF	Unordered
SEARCH_OVRFLW_SF	Unordered
DATA_SF	Unordered

(d)

Link	Linkset
I_j (for all j)	M:N list
S	1:N list with parent pointers
D	Singular pointer with parent pointer
R	1:N list
C	1:N doubly linked list

(e)

Fig. 33. The storage architecture of INQUIRE.

(i.e., ABSTRACT_DATA) records. For each PRIMARY_FRAG that is encountered in traversing link R, all of its associated SECONDARY_FRAG records are retrieved via link C. Adjoining the PRIMARY_FRAG record and its SECONDARY_FRAG records, and removing the delete flag and linkset fields, materializes a CONCEPTUAL record. By traversing link R in this manner, INQUIRE realizes a scan of a CONCEPTUAL file.

The internal files of INQUIRE are SIMPLE_INDEX, PREFIX_INDEX, SEARCH, SEARCH_OVERFLOW, PRIMARY_FRAG, and SECONDARY_FRAG. SIMPLE_INDEX and PREFIX_INDEX records are collectively organized by a single VSAM or ISAM file structure. SEARCH and SEARCH_OVERFLOW records are organized by separate BDAM or RSDS file structures.[9] PRIMARY_FRAG and SECONDARY_FRAG records are collectively organized by a single BDAM or RSDS file structure. These four file structures are called, respectively, the INDEX, SEARCH, SEARCH OVERFLOW, and the DATA files in INQUIRE documentation.

Figure 33 summarizes the storage architecture of INQUIRE. A data structure diagram that shows the levels of abstraction in INQUIRE and the elementary transformations that were applied to abstract files are presented in Figures 33a–b. Figure 33c gives the assignment of internal files to simple files, and Figures 33d–e list how each simple file and link is implemented.

This completes the derivation of INQUIRE's storage architecture. It is worth noting that our model of INQUIRE is quite accurate; the internal record types that were derived explain the presence and purpose of every pointer and every byte of the stored records that are documented in INQUIRE manuals. Source materials are [45, 46] and [25].

Finally, INQUIRE has support files, that were not considered in this derivation (e.g., ACCOUNTING and MACRO LIBRARY). These files could have been included in our model without much difficulty. Since their presence is optional and they do not constitute the core of INQUIRE's storage architecture, we ignored them for simplicity.

5. PERSPECTIVE, CONTRIBUTIONS, AND FUTURE WORK

There are three immediate contributions of our work: (1) The TM is the first model of physical databases capable of describing the internal structures of many operational DBMSs. Our research signals the beginning of a comprehensive reference to the storage architectures of popular DBMSs. Accurate descriptions of the architectures of commercially successful DBMSs should be quite valuable to future DBMS designers. (2) The TM provides a useful medium of communication. In just a few pages, the storage architecture of an actual or prototype DBMS can be conveyed in considerable detail and precision. Previously this was accomplished by reading cryptic (and often confusing) documentation and enormous software manuals. (3) Knowledge of the storage architectures of operational DBMSs ultimately improves one's understanding of database implementations in general.

[9] In UM terminology, VSAM is a B+ tree, ISAM is an indexed-sequential structure, BDAM is a one-level unordered file, and RSDS is a multileveled unordered file.

There are two long-term goals of our research: automated development of physical database software and performance and tuning packages for existing DBMSs. We address each in turn.

5.1 Automating the Development of Database System Software

Understanding the storage architecture of a DBMS is a necessary precondition to understanding the DBMS's behavior and performance. But it is not sufficient. Operations on files and links must also be considered. Elementary transformations have been explained in this paper as data mappings. Alternatively, they also could have been explained in terms of operation mappings (e.g., the mapping of record retrieval, insertion, deletion, and update operations). Consider, for example, the division transformation. The retrieval of an abstract record which has been divided involves a retrieval of the record's fragments followed by their concatenation; the insertion of an abstract record involves a division of the record, an insertion of the fragments, and a linking of the fragments. Materializations of update and deletion operations on abstract records are just as straightforward.

A central concept in understanding operation mappings is that the operations that are performed on abstract files and links are exactly the same as those that are performed on concrete files and links. That is, just as one can retrieve, insert, and delete conceptual records, so can retrievals, insertions, and deletions be performed on internal records. Thus the number of operations to be mapped is limited to the number of basic operations that can be performed on individual files and links, and this number is rather small [6]. It follows that for each basic operation and each transformation, a mapping is defined.[10] An attractive consequence of our model is that these mappings are valid for all levels of abstraction.

Elementary transformations describe how instances of an abstract record type, and operations on this type, are mapped to lower level types and operations. This is the basic idea of abstract data types [35]. In principle, one can define an abstract data type that encapsulates data and operation mappings for each transformation. As each data type supports exactly the same set of operations, they can be nested in many different ways. In other words, each abstract data type corresponds to a layer of software. Each layer has exactly the same interface. Layers can be stacked in different ways, so that the output of one layer becomes the input to the next.

As an example, consider the extraction and encoding transformations. Suppose a layer of software (abstract data type) exists for each. By stacking the extraction software on top of the encoding software, the output of the extraction layer becomes the input to the encoding layer. This would mean that nonencoded fields are indexed, and data records (and possibly index records) are later encoded. If the ordering of the layers were reversed, data records would be encoded first and then indices on encoded fields would be created.

From these ideas, it is not difficult to see that conceptual-to-internal mappings and elementary transformations are fundamental to the way DBMS software is

[10] Operation mappings may not be unique. For example, many different algorithms for searching transposed files have been proposed (see [4]). Each of these algorithms would define alternative mappings for retrieval operations.

actually written *or can be written.* Although it is fairly clear that existing DBMS software is not written in the highly structured and layered manner described above, it is nonetheless possible to identify some form of layering in real systems. We are presently developing a prototype system, called GENESIS, which is based on the TM approach [10]. The goal of the system is to automate the production of physical database software by routing the output of one layer of software to the input of another. In this way we hope to demonstrate that a technology for quickly developing special-purpose DBMS software is feasible. If the layers of software that a DBMS needs are already written, it is simply a matter of changing the routing tables to emulate the storage architecture of the DBMS. This can be done in a matter of hours; if the DBMS were built from scratch, its software development time would be measured in years.

We envision that our system can be used to produce DBMSs that emulate existing DBMSs and to produce DBMSs with hybrid architectures. These generated DBMSs can be used, for example, in simulation studies to determine what architectures are best for particular classes of applications.

Finally, we note that the value of abstract data types in database implementations has long been recognized [3, 37, 65]. However, the methods by which modular design concepts can be applied at the internal level are not well understood. We feel that our work can lead to an improvement and clarification of existing methods. Results on this subject are presented in [9] and [86].

5.2 Performance Prediction and Database Design Tools

Performance and design packages for commercial DBMSs can be developed once it is known how operations are mapped. The development of such packages will require the integration of performance prediction techniques with the descriptive techniques of the TM. Results using the UM and TM to design performance prediction tools for SYSTEM 2000 databases is forthcoming [17].

Tying performance prediction techniques to the TM does not mean that database optimization problems will be easier to solve; it simply means that the results of an optimization will be tailored to the peculiarities of a specific DBMS. For example, papers on index selection have used optimization models that were not tied to existing database systems. Thus it may be the case that the results of index selection for an INGRES database may be different (albeit slightly) from that of an ADABAS or INQUIRE database.

We believe that the TM provides a fresh perspective on some fundamental problems of physical database design. After reviewing a number of different storage architectures, it is natural to ask what is to be gained by using one transformation sequence rather than another. Clearly, such questions are significant, as they raise a fundamental point about what storage architectures (i.e., DBMSs) are better than others for given applications. No answer can yet be given. The present state of our research is to survey as many storage architectures as possible. Once sufficient knowledge has been collected, it is hoped that the underlying rules for generating and choosing transformation sequences will become evident. It is anticipated that the core of this research will center on an expert system for physical database design; design decisions would rely primarily on these rules and on results of simple performance calculations, rather than on the more traditional numerical optimization approaches.

Another reason for the need of additional surveys is that not all transformations are fully understood. For example, we noted in Section 2 that there is a tenth tranformation. It is commonly referred to as *horizontal partitioning* [2]. The basic idea is to partition a file of records into two or more groups. Differential files, [1] and [69], for example, partition records into two groups: modified and unmodified. Database machines [42] and distributed databases [18] also utilize horizontal partitioning. Unlike other elementary transformations, no explicit physical structures (e.g., delete flags, linksets) are added to horizontally partitioned files. However, metadata must be introduced in database schemas and algorithms to make such relationships explicit. Thus there appear to be transformations that introduce structure only at the schema level, not at the abstract and concrete data record levels. Additional research is needed to clarify these points.

6. CONCLUSIONS

Modeling the storage architecture of a DBMS is a prerequisite to understanding and optimizing database performance. Previously, such modeling was difficult because some fundamental principles of physical database design and implementation were not well understood. This has been clearly evident to researchers who have tried to use existing "general" models of physical databases to understand the internals of specific commercial DBMSs.

We have presented a model of conceptual-to-internal mappings, called the transformation model (TM), as an extension of the unifying model (UM) of Batory and Gotlieb. To place our work in context, we have shown (in Appendix I) that earlier models of physical databases are submodels of the UM. The domain of the UM is the implementation of internal files and links; simple files and linksets are the basic implementation constructs. The domain of the TM is the mapping of conceptual files and links to internal files and links; elementary transformations are the basic mapping constructs.

We have demonstrated that conceptual-to-internal mappings are fundamental to understanding physical database implementations. Elementary transformations provide the necessary means to express the complex storage architectures of operational DBMSs in a precise, systematic, and comprehendible way. We have outlined a relationship between elementary transformations and abstract data types, and their possible role in automating the production of internal DBMS software and in the development of performance packages for commercial DBMSs. We believe the transformation model is an important step toward tying physical database theory to practice.

APPENDIX I. Relationship with Earlier Work

We explain in this section how the unifying model (UM) provides a framework in which earlier models of physical databases can be cast. This explanation also serves as justification for using the UM as the starting point of our research. A familiarity with UM terminology is assumed. We also explain the relationship of earlier models with the TM.

The UM consists of two distinct submodels: one for simple files, the other for linksets. Most of the earlier general-purpose models are ancestors of the simple file submodel; DIAM [67] can be considered an ancestor of the linkset submodel. Some of our explanations are brief, as more elaborate discussions on historical lineages can be found in the cited papers.

The models of Hsiao and Harary [70] and Severance [69] were unified and extended by the access path model of Yao [88]. The simple file submodel of the UM is a direct extension of the access path model in that a more extensive parameterization of file structures was used. It is this parameterization that enabled different works and analyses on physical database design and performance to be related.

During the period when the above general-purpose models were being developed, important models of specific or restricted network databases were independently proposed by Das and Teorey [23], Mitoma and Irani [58], Gambino and Gerritsen [30], and Berelian and Irani [12], among others. The essential modeling constructs on which these works are based can be found in (or easily fitted into) the generalized UM framework, which is described in Section 2 and Appendix II. As an example, sequential and clustering linksets are discussed in [58] and [30]. Although the original UM did not accommodate these structures, the generalized framework does. In principle, the addition of more structures to the UM does not alter its framework; it simply enriches it.

More recently, March, Severance, and Wilens presented the frame memory model [55]. The *frame* was identified as a basic unit of physical database construction. The concept of a frame is identical to that of the UM concept of a node. The frame memory model concentrates on the implementation and selection of node formats while the UM does not. Again, it is not difficult to incorporate the frame memory model into the UM framework. The addition does not alter the framework; it simply enriches it.

The data independent accessing model (DIAM) was proposed in 1973 by Senko, Altman, Astrahan, and Fehder [67], and later extended by Fry, et al. [29]. Unlike other models, DIAM has not been directly related to subsequent modeling efforts. It is for the lack of historical connectivities that we devote a disproportionate part of our discussion to DIAM.

DIAM has "levels of abstraction" that foreshadow the three levels of the ANSI/SPARC proposal [81]. The only levels relevant to our discussions are the string and encoding levels.[11]

The basic modeling constructs of the string level are strings and atomic data values. An *atomic data value* is either a data value or the name of a string. A *string* is a sequence of atomic data values, with the first data value serving as the name of the string. Strings are used to form higher level constructs. For example, a string of atomic data values defines the concept of a "record," and a string of "records" defines a set of records. Sets of records can be collected onto strings to define higher level concepts such as indices (e.g., the cluster index of simple files). We call a set of atomic data values that are of the same type an *atomic value set*, and a set of homogeneous strings a *string set*. (We introduce these names because there are no corresponding terms in DIAM for their concepts.)

[11] The device level was used to describe the physical characteristics of secondary storage devices; such descriptions are independent of the descriptions of the simple files and linksets that may be stored on them. That is, one can model hash-based files, indexed-sequential files, pointer arrays, and so on without ever having to define specifically their storage medium (e.g., floppy, drum, disk). In fact, almost all results on database performance since 1977 (in particular, query optimization and database design) have avoided such details. We concur with this trend.

DIAM and the UM have a straightforward correspondence. An atomic data value in DIAM corresponds to a record (with a single field) in the UM. Strings are relationships between atomic data values in DIAM; they are link occurrences in the UM. DIAM atomic value sets correspond to UM files, and DIAM string sets correspond to UM links. In this way both DIAM and the UM can use data structure diagrams to represent relationships among files (atomic value sets).

The implementation of string sets is specified at the encoding level of DIAM. In the original paper, strings could be implemented by lists or by sequential linksets. The extension to DIAM by Fry, et al. introduced pointer array linksets.[12] However, there is no provision in DIAM or in its extension that treats simple file structures as primitive constructs, or accounts for the multitude of variations that can accompany list, sequential, and pointer array linksets. It is for this reason that DIAM can be considered an ancestor of the linkset submodel of the UM. A more detailed connection between DIAM and the UM is given in [8].

In summary, earlier models of physical databases are submodels of the UM. The UM does *not* show how simple file structures and linkset structures are related to conceptual-to-internal mappings or how DBMS software transforms conceptual records into internal records in a stepwise fashion. These are the tasks that are handled by the TM. (Note that the TM does *not* introduce new simple file and linkset structures; the structures that the TM uses are those provided by the UM.)

APPENDIX II. Catalogs of Recognized Simple Files and Linksets

Simple Files

Simple files have a common description: they can be modeled as uniform-height directed trees where the vertices of a tree correspond to the standard notions of secondary storage *nodes* or *frames*. There are, however, fundamental differences among simple file types. The major differences can be delineated with the aid of four parameters: CK, GROWTH, ACCESS, and SEQUENCING. In the following we assume a minimal familiarity with the UM terminology.

Parameter 1. CK (cluster key type). A simple file organizes internal records according to a single key called the *cluster key*.[13] Three types of cluster keys are known: (1) A *logical-valued key* is a key that is contained in internal records. B+ trees, sequential, and indexed-sequential structures use logical-valued keys. (2) A *hash key* is an algebraic transformation of a logical-valued key. Hash-based

[12] A basic premise of DIAM is that lists are the fundamental string implementation. To explain the existence of other methods, factoring and embedding were introduced. Factoring [67] is simply a mapping from list linksets to sequential linksets. Embedding [29] is a mapping of lists to pointer arrays. Thus other linkset implementations were to be viewed as derivatives of list linksets. If the UM approach were taken where several fundamental string implementations are recognized, factoring and embedding could be eliminated as artificial constructs.

[13] There are simple files that organize records on several keys. See [62] for an example and survey.

Table II. A Catalog of Simple Files

Simple file	CK	Growth	Access	Sequencing	Comments
Indexed-aggregate	Logical-valued	Overflow	Random	Unordered	[39]
Indexed-sequential	Logical-valued	Overflow	Random	Ordered	[13]
B+ tree	Logical-valued	Splitting	Random	Ordered	[21]
Sequential	Logical-valued	Locational	Sequential	Ordered	[85]
Deferred B+ tree	Logical-valued	Deferred	Random	Ordered	[59, 88]
Hash-based	Hash	Overflow	Random	Unordered	[5]
Dynamic hash-based	Hash	Splitting	Random	Unordered	[26, 51]
Deferred hash-based	Hash	Deferred	Random	Unordered	[66]
Linear hash-based	Hash	Linear	Random	Unordered	[66]
Unordered	Relative	Locational	Random	Unordered	[85]
Heap	Relative	Locational	Sequential	Unordered	[76]
B-list	Relative	Splitting	Random	Unordered	[78]

and dynamic hash-based structures organize records on hash keys. (3) A *relative key* specifies an internal record's index position relative to the start of the file (e.g., the *i*th record of the file). Unordered and heap files use relative keys.

Parameter 2. GROWTH (method of file growth). A simple file can accommodate file growth in one of five basic ways. (1) *Overflow*—new records are placed on overflow chains. Hash-based and indexed-sequential files use overflow. (2) *Splitting*—nodes are split when they "overflow." B+ trees and dynamic hash-based files [26, 51] use node splitting. (3) *Locational*—new records are inserted wherever there is room, usually at the end of a file. Unordered and heap file structures are examples. (4) *Deferred splitting*—a generalization of node splitting. Instead of splitting a node when it is about to overflow, node splitting is triggered after a certain amount of overflow has occurred [59, 66, 88]. (5) *Linear splitting*—a generalization of overflow. Nodes are split in a predetermined sequence, and splitting is triggered in order to maintain a constant loading factor [66].

Parameter 3. ACCESS (random or sequential access). The primary purpose of the cluster index for most simple files is to facilitate the fast retrieval of internal records, given their cluster keys. If this is the case, *random* accessing of records is possible, otherwise only *sequential* accessing of internal records can be performed.

Parameter 4. SEQUENCING (ordering of records). Records are either maintained in an *unordered* sequence or they are *ordered* in ascending or descending logical-valued key sequence.

A spectrum of simple files is defined by taking combinations of different parameter values (see Table II). Many combinations can be readily identified with known structures, but not all describe implementations that are meaningful. However, there are some combinations that cannot be ruled out and cannot be identified with recognized structures. One is an indexed-sequential file that uses linear splitting to accommodate file growth. Such a structure would appear to have the properties of indexed-sequential files, with the important difference that it, like linear hash-based files, does not require periodic reorganization. This structure has yet to be studied in detail.

Linksets

A link is a generalization of the CODASYL set. Every link has precisely one *parent file* and one or more *child files*. It is possible for a file to assume the role of *both* parent and child in a link. The basic unit of connectivity is the *link occurrence*, which consists of one parent record and the zero or more child records to which it is related. It is possible for a child record to participate in *many* occurrences of the same link at the same time, and thus have multiple parent records. Therefore, links can represent 1:1, 1:*N*, and *M*:*N* relationships.

Every parent record and every child record has a *link key*. A link key can be an explicit part of a record or it can be inferred. (If it is inferred, the link is said to be *information carrying* [80]). A link occurrence consists of a parent record and all child records that have the same link key as the parent. It is usually the case that link keys for parent records are identifiers (primary keys) and link keys for child records are nonidentifiers. Thus most links are 1:*N*. *M*:*N* links arise when either parent records, child records, or both have repeating groups as link keys. ADABAS and INQUIRE support *M*:*N* links.

Four fundamental types of link implementations have been recognized to date: serial, list, sequential, and relational. *Serial* linksets connect parent records to child records by pointer arrays, *list* linksets make connections by list structures, *sequential* linksets have connections based on physical locality, and *relational* linksets rely on file searching (for records that have the same link key). Serial, list, and relational linksets can be used to implement *N*:*M* links. Sequential linksets can only implement 1:*N* links. Figure 34 illustrates their basic differences.

An implementation option common to all linkset types is the presence of *parent pointers* (i.e., pointers from child records to parent records). List linksets and serial linksets have a number of additional options. For list linksets, a "list" can be a linear list or a ring list. It can be doubly-linked. There can also be a pointer (stored with the parent record) to the last child record of an occurrence. Each variation has been used in one or more DBMS implementations.[14]

A pointer array of a serial linkset is a repeating group, where the repeating unit is the address of (i.e., a pointer to) a child record. Optionally, some of the data values of a child record *in addition* to its address can be the repeating unit. In such cases serial linksets are said to be *keyed*. Figure 35 shows two keyed serial linkset occurrences where the repeating unit is data fields *B* and *C* and a pointer. SPIRES [74] uses keyed serial linksets as generalizations of inverted lists to enhance secondary key retrieval of data records.[15]

List and serial linksets have two variations in common: clustering and cellular. When child records are stored near their parent records, the linkset is said to be *clustered*. Clustering is restricted to 1:*N* links. IDMS and DMS-1100, for example, implement CODASYL sets by ring lists or pointer arrays. Child records can be clustered about their parent records with the LOCATION MODE IS VIA schema declaration.

List and serial linksets can exploit a partitioning of the child file(s) into subfiles called *cells*. A cell contains an integral number of nodes. With respect to

[14] Pointers can be either physical addresses or symbolic keys. Physical pointers are preferred if the storage location of internal records always remains constant.

[15] A variant of this approach, where hash values are stored, is described in [47].

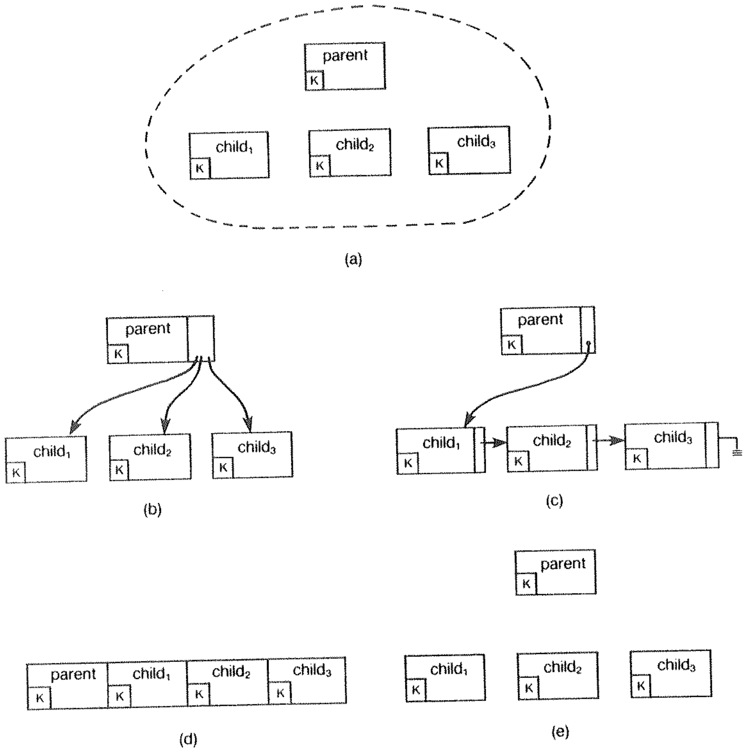


Fig. 34. The basic linkset types. (a) A link occurrence with link key K. (b) A serial linkset occurrence. (c) A list linkset occurrence. (d) A sequential linkset occurrence. (e) A relational linkset occurrence.

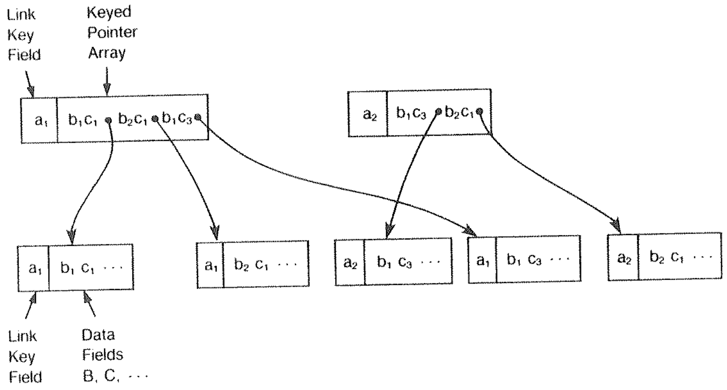


Fig. 35. Two keyed serial linkset occurrences.

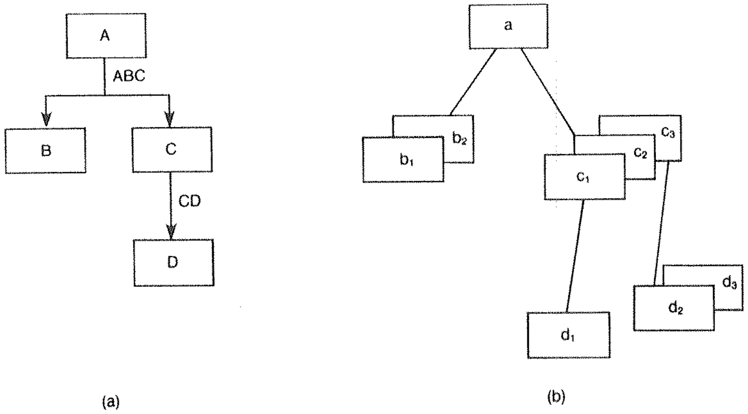


Fig. 36. A tree data structure diagram and a tree occurrence.

a given link occurrence, a cell is said to be *occupied* if it contains a child record of the link occurrence. Otherwise it is unoccupied.

Each pointer of a cellular serial pointer array references a distinct cell that is occupied; it identifies the starting address of the cell. Thus, to locate child records requires a scan of the cell. ADABAS uses cellular serial linksets with cells that contain precisely one block.

Cellular list linksets (sometimes referred to as *cellular multilists*) also use pointer arrays. Each pointer identifies the head of a list of child records (of a link occurrence) that are stored in the same cell. Thus the number of pointers in a cellular list pointer array is the number of occupied cells for the corresponding link occurrence. No commercial DBMS, to the author's knowledge, uses cellular multilists, even though this linkset has often been discussed in the literature.

Two major variations of linksets are hierarchical and record sequencing. Linksets usually implement one link, but they can also realize two or more links. In these cases the data structure diagrams of the links and their attendant parent and child files are required to form a tree. Figure 36a shows a tree data structure diagram. An instance of the tree (which consists of a record of the root file and all of its descendants) is a *tree occurrence.* Figure 36b shows a typical tree occurrence.

An occurrence of a *hierarchical* linkset is a tree occurrence which has been flattened into a two-level hierarchy. The parent–child relationships of the tree occurrence are preserved by arranging descendant records in *hierarchical sequence.* That is, the tree occurrence is traversed in preorder traversal (visit the root, visit in left-to-right order the subtrees headed by each of its child records) to linearize the descendant records. The root record of a hierarchical linkset assumes the role of "parent" and its descendant records assume the role of "children." This flattening enables sequential, list, and serial linksets to be used to implement hierarchical linksets. Figure 37 illustrates their differences. IMS

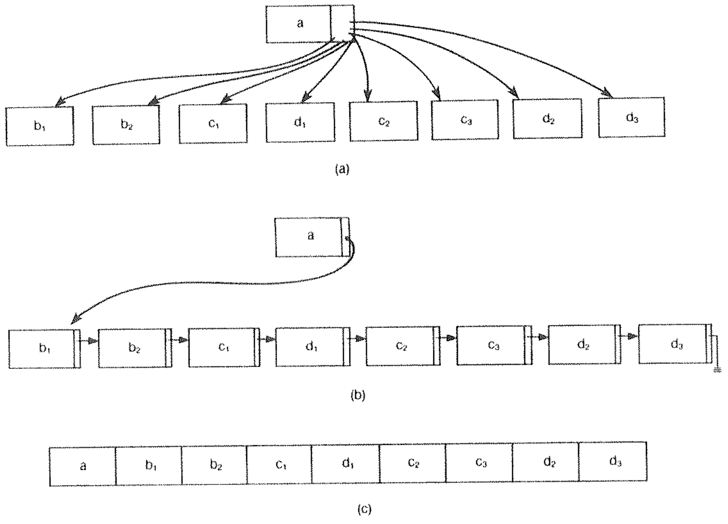


Fig. 37. Basic hierarchical linkset types.

uses hierarchical, sequential, and hierarchical list linksets. It is not known if any DBMS uses hierarchical serial linksets.

The second major variation of linksets is the ordering of child records. If a link has but one child file, the child records of a linkset occurrence can be arranged in a user-defined order, in a random order, or in an ascending or descending chronological, physical address, or sortkey order. When a link has two or more child files, one of three different options must also be specified: sorted, grouped, or ungrouped.

Consider a linkset that implements a single link that has two or more child types. If all child types have a sort field in common, then their instances can be *sorted* in ascending or descending sortkey order. Alternatively, orderings can be separately imposed on the records of each child type. Thus, if a link has two child files, an occurrence would consist of a parent record and two sequences of child records, one for each type. If the linkset maintains the concatenation of both sequences, child records are said to be *grouped*. (The ordering of the sequences is determined by the left-to-right appearance of the child files in the underlying data structure diagram.) Child records are *ungrouped* if the linkset just maintains the relative ordering of records within each type, thereby allowing records of different sequences to be interleaved. Figure 38 illustrates the basic differences among sorted, grouped, and ungrouped. DMS-1100, SYSTEM 2000, and IDMS support the ungrouped option. DMS-1100 additionally supports sorted. IMS uses the grouped option.

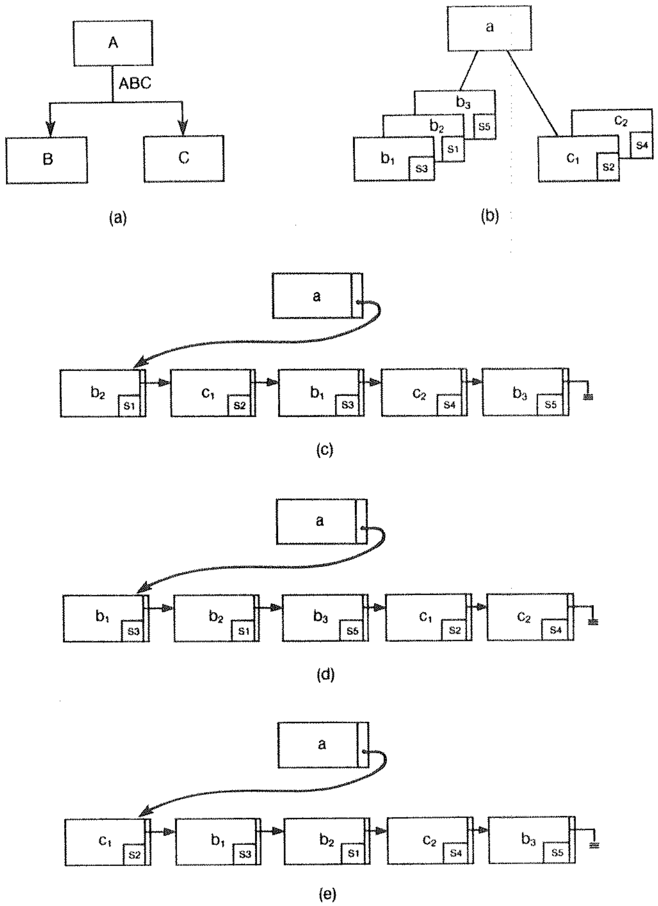


Fig. 38. Sorted, grouped, and ungrouped linkset occurrences.

The notions of sorted, grouped, and ungrouped generalize to hierarchical linksets in a natural way. Sorting and grouping rules are applied to the child record types of each parent in a tree data structure diagram. Figure 37 illustrates IMS's hierarchical sequential and list linksets with the grouped option. It is not known whether any DBMS uses ungrouped or sorted hierarchical linksets.

It is evident from the above discussions that there is a combinatorial number of linkset implementations. A provisional naming scheme has been devised that

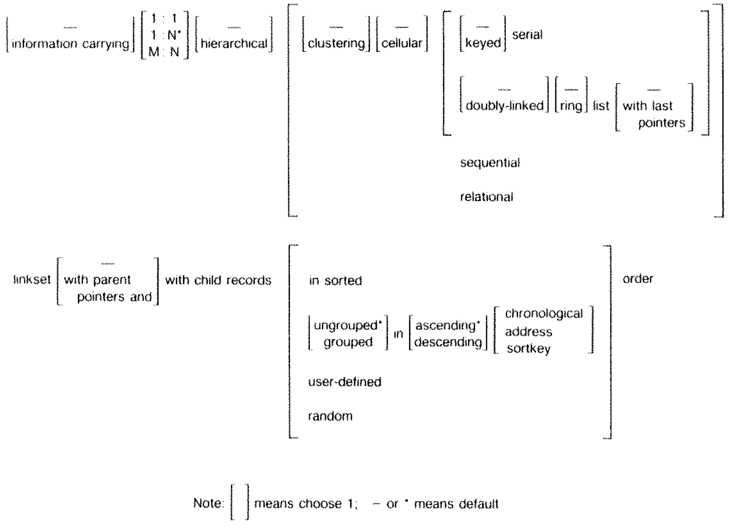


Fig. 39. A classification of linksets.

Table III. Common Linkset Names and Their Definitions

Common name	Description
Multilist	A list linkset that is not information-carrying, hierarchical, or cellular; child records are usually kept in chronological or address order.
Inverted list	A serial linkset that is not information-carrying, hierarchical, or cellular; child records are usually kept in chronological or address order.
Pointer array	A serial linkset that is not hierarchical.
DBTG ring list	A ring list linkset that is not hierarchical or cellular.
Singular pointer	A 1 : 1 serial linkset, usually information-carrying.
IMS hierarchical pointers	A hierarchical list linkset (possibly doubly linked) with child records grouped.
IMS child/twin pointers	A list or doubly linked list linkset.
IMS logical parent pointers	A information-carrying relational linkset with parent pointers.
Transposed	A 1 : 1 sequential linkset with parent and child records stored in separate unordered files.
Index encoded	An information-carrying relational linkset with parent pointers. Parent records are stored in an unordered file.

enables recognized linkset implementations to be classified. This scheme is given in Figure 39. Note that not all combinations have been or can be implemented. Many of the exceptions have already been noted. Furthermore, some linksets are so common they are given special names—they are listed in Table III.

APPENDIX III. ADABAS

ADABAS is a product of Software AG, Inc. A typical ADABAS database is populated with one or more conceptual files which may be related explicitly by *couplings* or implicitly by join operations. A representative ADABAS data structure diagram is shown in Figure 40. Couplings are represented by bidirectional links that connect two different conceptual files. ADABAS does not allow for a file to be coupled with itself, or for more than one coupling to exist between two files at any one time.[16]

The generic CONCEPTUAL record type supported by ADABAS consists of n fields, $F_1 \dots F_n$, which are elementary or compound. An elementary or compound field may be scalar or repeating. Data values can have variable lengths. Generally, CONCEPTUAL records are variable length.

A coupling between records is made by sharing a common value in designated fields. Because fields may be repeating, couplings can be M:N. Figure 41.id illustrates an M:N coupling.

The internal files and links of ADABAS are derived in the following way. Each coupling is *actualized* by a pair of oppositely-directed internal links; both links are realized by M:N pointer arrays.[17] The pointers of each array are maintained in order of ascending addresses. Figure 41 shows the actualization of the coupling between the CONCEPTUAL$_i$ and CONCEPTUAL$_j$ files. Two ABSTRACT_CONCEPTUAL files and two internal links are produced in the process. An actualization of the couplings in the database of Figure 40 would produce a total of four ABSTRACT_CONCEPTUAL files and eight internal links.

The generic form of an ABSTRACT_CONCEPTUAL record is shown in Figure 42. An ABSTRACT_CONCEPTUAL record is the parent of m links $L_1 \dots L_m$ which were produced by the actualization of m couplings. A record consists of data fields $F_1 \dots F_n$ and m parent fields $P_{L_1} \dots P_{L_m}$.

ABSTRACT_CONCEPTUAL records are materialized in two steps (see Figure 43). First, fields $P_{L_1} \dots P_{L_m}$ are individually *segmented* from the data fields $F_1 \dots F_n$. The result is $m + 1$ files and m links: there is an ABSTRACT_DATA file (containing only data fields), and for each parent field P_{L_k} there is an ABSTRACT_ASSOCIATOR$_k$ file connected to ABSTRACT_DATA by link A_k. A_k is realized by a singular pointer.

Second, ADABAS allows scalar and repeating fields to be indexed. Field F_j is indexed by *extracting* it from ABSTRACT_DATA. This creates an

[16] Couplings are used in only 1–2 percent of ADABAS databases because their utility is limited to processing specialized queries and because they degrade performance significantly for update-intensive files [32]. Couplings are supported in the most recent release of ADABAS, but their use is not recommended. Join operations are promoted instead.

[17] It is also correct to say that a coupling is actualized by a single link whose implementation is an M:N pointer array with parent pointers. This interpretation, however, forces one record type to be arbitrarily labeled as the "parent" and the other as the "child." This results in a more complicated, but equivalent, derivation.

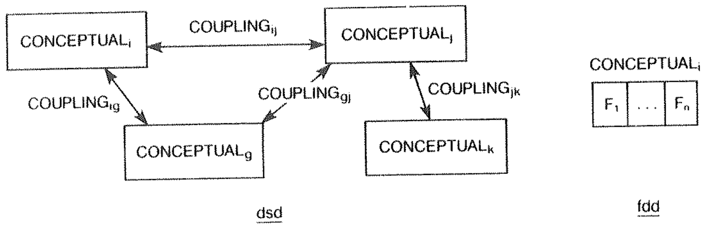


Fig. 40. A representative ADABAS dsd and fdd.

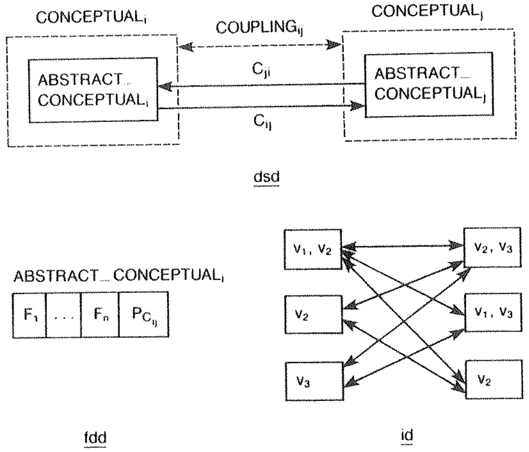


Fig. 41. Actualization of a coupling.

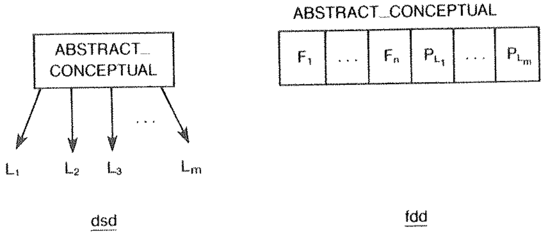


Fig. 42. Generic ABSTRACT_CONCEPTUAL record type.

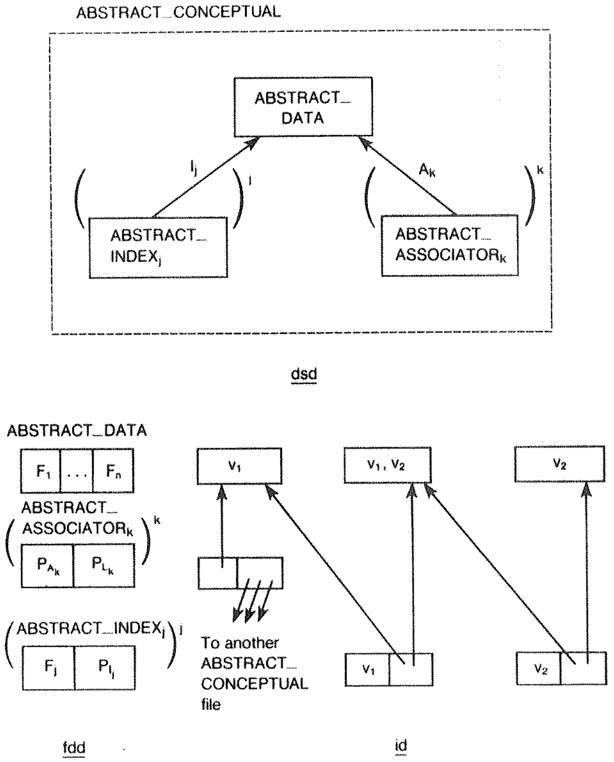


Fig. 43. Segmentation and extraction of ABSTRACT_CONCEPTUAL records.

ABSTRACT_INDEX$_j$ file. Link I_j, which connects ABSTRACT_INDEX$_j$ to ABSTRACT_DATA, is realized by an $M{:}N$ inverted list (i.e., an $M{:}N$ pointer array). The pointers of each inverted list are maintained in order of ascending addresses. All fields are indexed in this manner.

Note that if a CONCEPTUAL file was uncoupled and had no indexed fields, it would be mapped directly to an ABSTRACT_DATA record via the *null* transform.

Figure 43.id illustrates the relationships among three ABSTRACT_DATA records, two ABSTRACT_INDEX records, and one ABSTRACT_ASSOCIATOR record. Note that only the contents of a single repeating field of each ABSTRACT_DATA record is shown; this field contains value v_1 in one record, values v_1 and v_2 in a second, and value v_2 in a third.

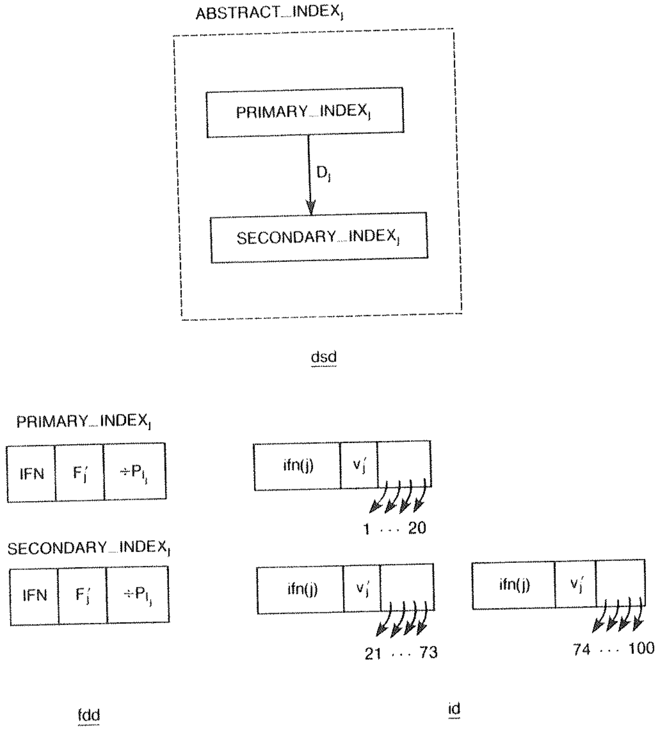


Fig. 44. Augmentation, encoding, and division of ABSTRACT_INDEX$_j$ records.

Pointers to ABSTRACT_DATA records are known as *internal sequence numbers* (ISNs). A distinct ISN is assigned to each CONCEPTUAL record and is used to locate the record. Its realization is explained later. *Internal file numbers* and *internal field numbers*, which we collectively call IFNs, are used internally by ADABAS to reference CONCEPTUAL files and their constitutent fields. Field numbers are distinguishable from file numbers.

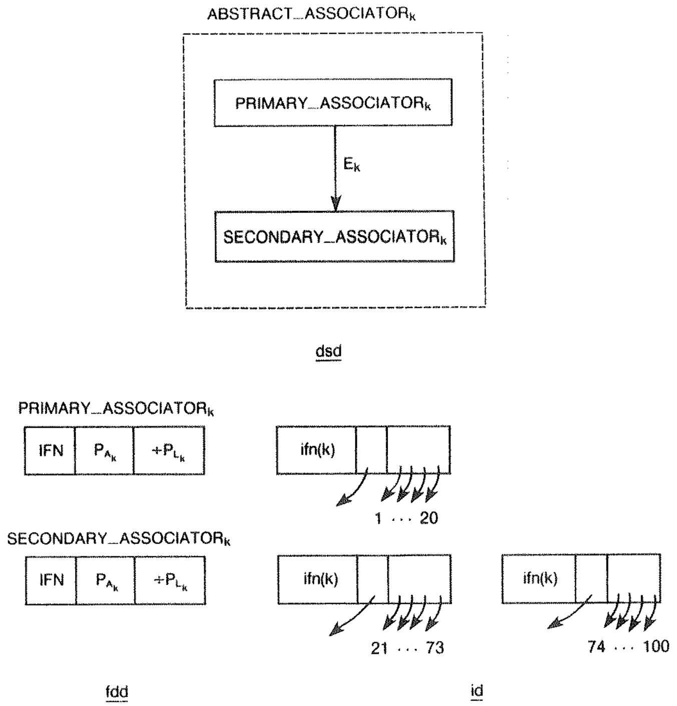


Fig. 45. Augmentation and division of ABSTRACT_ASSOCIATOR$_k$ records.

An ABSTRACT_INDEX$_j$ record is materialized in three steps (see Figure 44). First, a field containing the IFN of field F_j is *augmented*. Second, the value in field F_j is *encoded* by an ADABAS compression technique (see [32]). The encoded field is labeled F_j' in Figure 44.fdd. Third, the record may be *divided* into one or more fragments with the IFN and F_j' fields duplicated in each fragment. The first fragment is of type PRIMARY_INDEX$_j$ and the remaining are of type SECONDARY_INDEX$_j$. The fragment files are connected by link D$_j$, which is realized by a relational linkset with link key (IFN, F_j'). (The conditions under which division occurs will be explained shortly.) Figure 44.id shows how an ABSTRACT_INDEX$_j$ record with an inverted list of 100 pointers might be divided into three fragments. (v_j' is an encoded data value and ifn(j) is the IFN for field F_j). PRIMARY_INDEX$_j$ and SECONDARY_INDEX$_j$ are internal files.

An ABSTRACT_ASSOCIATOR$_k$ record is materialized in a similar manner (see Figure 45). First, a field containing the IFN of the child file of link L_k is *augmented*. Second, the record is *divided* into one or more fragments with the IFN and P_{A_k} fields duplicated in each fragment. The first fragment is of type PRIMARY_ASSOCIATOR$_k$ and the remaining are of type SECONDARY_ASSOCIATOR$_k$. The fragment files are connected by link E_k, which is implemented by a relational linkset with link key (IFN, P_{A_k}). Figure 45.id shows how an ABSTRACT_ASSOCIATOR$_k$ record with a pointer array of 100 pointers might be divided into three fragments. PRIMARY_ASSOCIATOR$_k$ and SECONDARY_ASSOCIATOR$_k$ are internal files.

ADABAS forces records of all PRIMARY_INDEX, SECONDARY_INDEX, PRIMARY_ASSOCIATOR, and SECONDARY_ASSOCIATOR types to have a similar format so that they can all be organized by a single file structure rather than having a separate file structure for each type. The file structure used is a

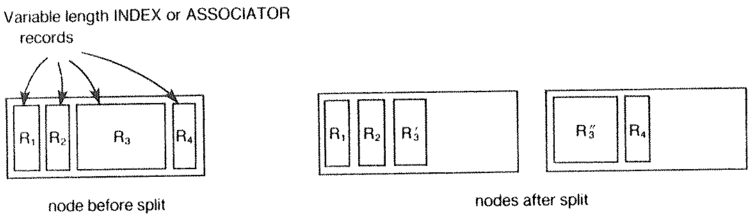


Fig. 46. Illustration of dividing ABSTRACT_INDEX and ABSTRACT_ASSOCIATOR records.

B+ trie, which is similar to B+ trees, in that file growth is accommodated by node splitting.[18] The division of an ABSTRACT_INDEX and ABSTRACT_ASSOCIATOR record is a result of node splitting. When a node splits, two nodes are created; both are approximately half full. Although ABSTRACT_INDEX and ABSTRACT_ASSOCIATOR records are variable length, loading both nodes equally is not a difficult task if the records are much smaller than the size of a node. When records are large, however, loading both nodes evenly is not possible without dividing one record into two and storing them in different nodes. Figure 46 illustrates the splitting of a node and the division of record R3 into R3′ and R3″.

The ABSTRACT_DATA file of Figure 43 is materialized in two steps. First, all data fields are *encoded* by an ADABAS compression technique. Second, the *indirection* transformation is applied. What results is an ADDRESS_CONVERTER file and a COMPRESSED_DATA file connected by link *AC*. An ADDRESS_CONVERTER record has a fixed length and contains only the field P_{AC}; a COMPRESSED_DATA record has a variable length and contains compressed data fields $F_1' \ldots F_n'$ and field C_{AC}. Link *AC* is realized by a pointer to the block that contains the associated COMPRESSED_DATA record, and the COMPRESSED_DATA record has a pointer back to its ADDRESS_CONVERTER record (Figure 47). This is a cellular singular pointer with a parent pointer.

Note that the ADDRESS_CONVERTER records maintain the 1:1 correspondence between ISNs and the storage locations of COMPRESSED_DATA records. Because of this correspondence, a COMPRESSED_DATA record can be relocated in secondary storage without altering the inverted lists and pointer arrays of ABSTRACT_INDEX and ABSTRACT_ASSOCIATOR records that reference it. (The pointers of these lists and arrays are ISNs). Relocations occur when there is no room in a block to accommodate an expanded COMPRESSED_DATA record. Expansions happen when a CONCEPTUAL record is modified, such as adding a new value to a repeating field.

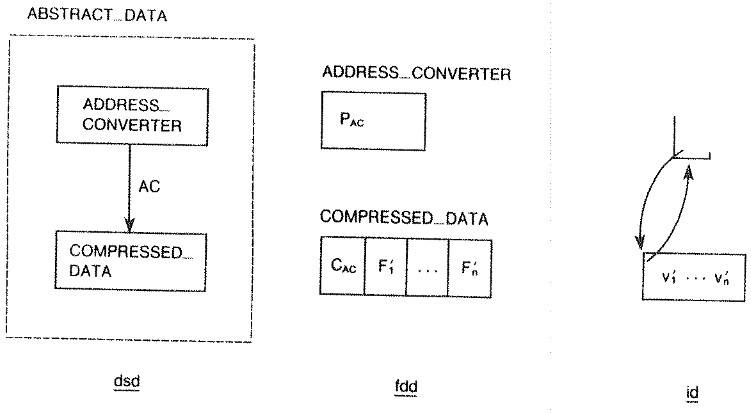


Fig. 47. Segmentation and encoding of ABSTRACT_DATA records.

The internal files of ADABAS are PRIMARY_INDEX$_j$, SECONDARY_INDEX$_j$, PRIMARY_ASSOCIATOR$_k$, SECONDARY_ASSOCIATOR$_k$, COMPRESSED_DATA, and ADDRESS_CONVERTER. Every CONCEPTUAL file is materialized by a collection of these files and each collection is organized by a separate group of file structures. For each CONCEPTUAL file, all record occurrences of all PRIMARY_INDEX$_j$, SECONDARY_INDEX$_j$, PRIMARY_ASSOCIATOR$_k$, and SECONDARY_ASSOCIATOR$_k$ files are organized by a single B+ trie (see [32, 50] and footnote 18 of this Appendix). The ADDRESS_CONVERTER file is organized by an unordered file structure and the COMPRESSED_DATA file is organized by a heap. An ISN is the relative key of an ADDRESS_CONVERTER record.

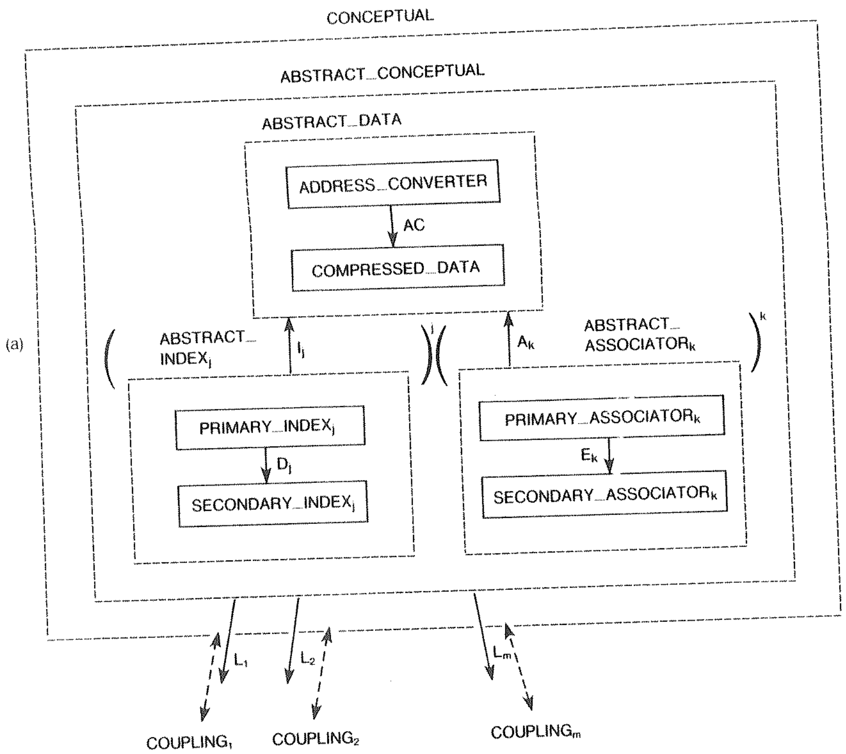
ADABAS places all B+ tries and ADDRESS_CONVERTER file structures that belong to a single database in an area of secondary storage called the "associator." (This is not to be confused with the ASSOCIATOR record types.) The COMPRESSED_DATA files of the database are placed in another called "data storage." Separate "associator" and "data storage" areas exist for different databases.

Figure 48 summarizes the storage architecture of ADABAS. Source materials are [32, 50, 71, 72] and [87].

APPENDIX IV. SYSTEM 2000

SYSTEM 2000 is a product of MRI Systems Corporation (now Intel). SYSTEM 2000 organizes conceptual files according to a hierarchical data model. A database is viewed as a collection of disjoint trees that have record occurrences as vertices. Each tree is referred to as a *database tree* and consists of one root record and all

[18] A B+ trie is a hybridization of the trie [28, 79] and the B+ tree. The B+ trie used in the most recent release of ADABAS has from one to six levels. The top levels partition records on their IFN and F_j or P_{A_k} values. The second lowest level partitions records on F_j or P_{A_k} and ISN values. The bottom level contains the PRIMARY_INDEX, SECONDARY_INDEX, PRIMARY_ASSOCIATOR, and SECONDARY_ASSOCIATOR records.



Abstract File	Elementary Transformations
CONCEPTUAL	Actualization
ABSTRACT CONCEPTUAL	Segmentation and extraction
ABSTRACT INDEX$_j$ (for all j)	Augmentation, encoding, and division
ABSTRACT ASSOCIATOR$_k$ (for all k)	Augmentation and division
ABSTRACT DATA	Encoding and indirection

(b)

Internal File	Simple File
PRIMARY INDEX$_j$ (for all j)	INDX SF
SECONDARY INDEX$_j$ (for all j)	INDX SF
PRIMARY ASSOCIATOR$_k$ (for all k)	INDX SF
SECONDARY ASSOCIATOR$_k$ (for all k)	INDX SF
ADDRESS CONVERTER	ADDR CNVTR SF
COMPRESSED DATA	DATA SF

(c)

Simple File	Implementation
INDX SF	B+ tree
ADDR CNVTR SF	Unordered
DATA SF	Heap

(d)

Link	Linkset
I$_j$ (for all j)	N : M pointer array with child records in ascending physical address order
L$_m$ (for all m)	N : M pointer array with child records in ascending physical address order
A$_k$ (for all k)	Singular pointer
AC	Cellular singular pointer with parent pointer
D$_j$ (for all j)	Relational linkset with child records in sortkey order
E$_k$ (for all k)	Relational linkset with child records in sortkey order

(e)

Fig. 48. The storage architecture of ADABAS.

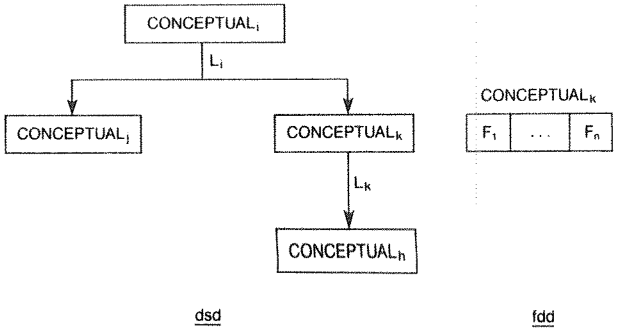


Fig. 49. A representative SYSTEM 2000 dsd and fdd.

of its dependent records. All database trees are instances of a *hierarchical definition tree* which specifies the hierarchical relationships among conceptual files.[19] A definition tree allows the parent, children, ancestors, and descendents of a record to be identified in a natural way. A representative SYSTEM 2000 definition tree is shown in Figure 49.

The generic CONCEPTUAL record type supported by SYSTEM 2000 consists of n data fields, $F_1 \ldots F_n$, which are elementary and scalar. Nonnumeric data values may have variable lengths; numeric values have fixed lengths. Generally, CONCEPTUAL records are variable-length.

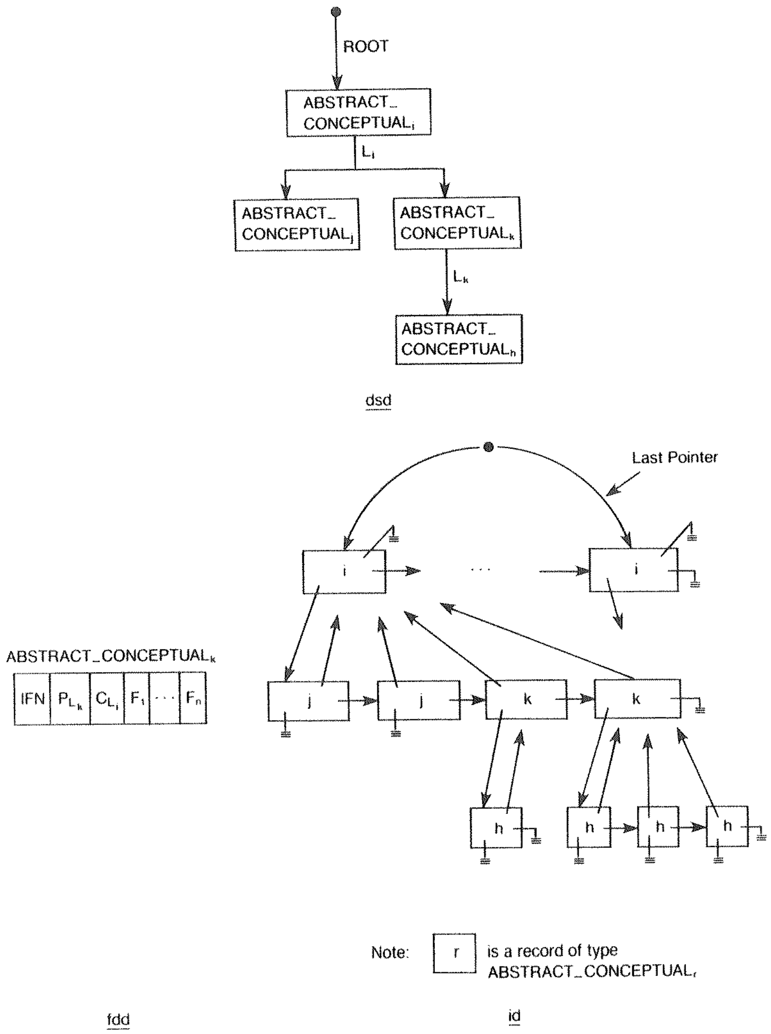
The first step in the materialization of a hierarchical definition tree is to transform CONCEPTUAL records into ABSTRACT_CONCEPTUAL records by specifying the implementation of the links in the hierarchical definition tree. ABSTRACT_CONCEPTUAL records, it turns out, are fairly easy to understand, but their derivation is rather complicated. To make the derivation comprehendible, we first describe an ABSTRACT_CONCEPTUAL record.

An ABSTRACT_CONCEPTUAL record differs from its CONCEPTUAL record counterpart by the addition of three fields (Fig. 49.ffd and Fig. 51.fdd). One field, labeled IFN, identifies the CONCEPTUAL file. A second, labeled P_D, is a parent field which contains a pointer to the first child record of a link D occurrence. A third, labeled C_A, is a child field which contains a parent pointer and a pointer to the next child of a link A occurrence. These fields are introduced as a result of the following four step derivation (see Figure 50).

(1) SYSTEM 2000 distinguishes different CONCEPTUAL files by assigning them distinct internal file numbers (IFNs). Each CONCEPTUAL record is *augmented* with a field containing its respective IFN.

(2) The link between a parent file and all of its immediate child files in a hierarchical definition tree is realized by a single linkset where the roles of parent and child are preserved. The linkset is a list with parent pointers. Child records

[19] Terms such as record type, database tree, and hierarchical definition tree are taken from Tsichritzis and Lochovsky [80]. Different releases of SYSTEM 2000 have used different sets of terminology.



dsd

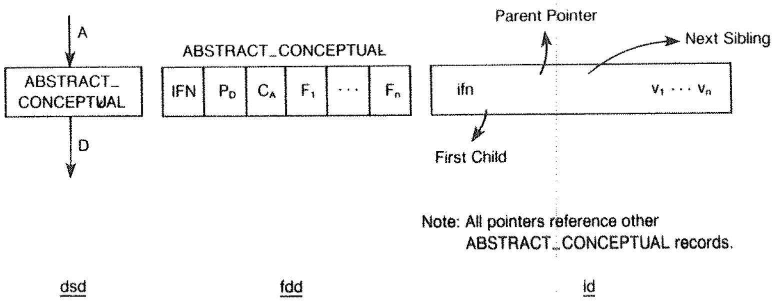
fdd

id

Fig. 50. Augmentation and collection of CONCEPTUAL records.

are arranged in user-defined order. Figure 50.id illustrates a possible arrangement. All links in a hierarchical definition tree are realized in this manner.

Observe that assigning list implementations to each conceptual link introduces a parent field in the root record type, a child field in leaf record types, and both parent and child fields in the intermediate record types of a hierarchical definition tree. In order for all records of all ABSTRACT_CONCEPTUAL files to have both parent and child fields, some null pointer fields must be introduced. This is done in the remaining two steps.



dsd

fdd

id

Fig. 51. Generic ABSTRACT_CONCEPTUAL record type.

(3) So that all instances of the root record type can be assessed efficiently, root records are *collected* together by link ROOT. ROOT is implemented as a list linkset (with precisely one occurrence) with parent pointers and a pointer to the last root record.[20] (Note that a parent pointer to the system • record is indistinguishable from a null pointer.) Root records are arranged in a user-defined order.

(4) A field containing a single null pointer is *augmented* to each leaf record type of a hierarchical definition tree. This field is indistinguishable from a parent field (labeled P_D in Figure 51.fdd) of a list linkset where there are no child records. These null pointers are shown in the occurrence of record types ABSTRACT_CONCEPTUAL$_j$ and ABSTRACT_CONCEPTUAL$_h$ in Figure 50.id.

The generic form of an ABSTRACT_CONCEPTUAL record is shown in Figure 51. An ABSTRACT_CONCEPTUAL record is a child of link A (A for ancestor) and is the parent of link D (D for descendent). (Note that a specific instance of A is ROOT.) An ABSTRACT_CONCEPTUAL record consists of an IFN field, a parent field P_D, a child field C_A, and n data fields $F_1 \ldots F_n$.

An ABSTRACT_CONCEPTUAL record is materialized in the following way (see Figure 52). SYSTEM 2000 creates an index for all data fields, unless told otherwise in the schema definition. Field F_j is indexed by *extracting* it from ABSTRACT_CONCEPTUAL records, forming an ABSTRACT_INDEX$_j$ file. Link I_j, which connects ABSTRACT_INDEX$_j$ to ABSTRACT_DATA, is implemented by a $1{:}N$ inverted list. Pointers of an inverted list are in chronological order. Other fields are indexed in an identical manner.

Note that if an ABSTRACT_CONCEPTUAL file had no indexed fields, it would be mapped directly to the ABSTRACT_DATA file via the *null* transformation.

[20] SYSTEM 2000 actually stores the pointer to the last root record in the parent pointer slot of the first root record of the ROOT list. (Normally, this slot would otherwise be occupied by a null pointer.) A slightly more efficient implementation would store the last pointer in the system • record as shown in Figure 50.id.

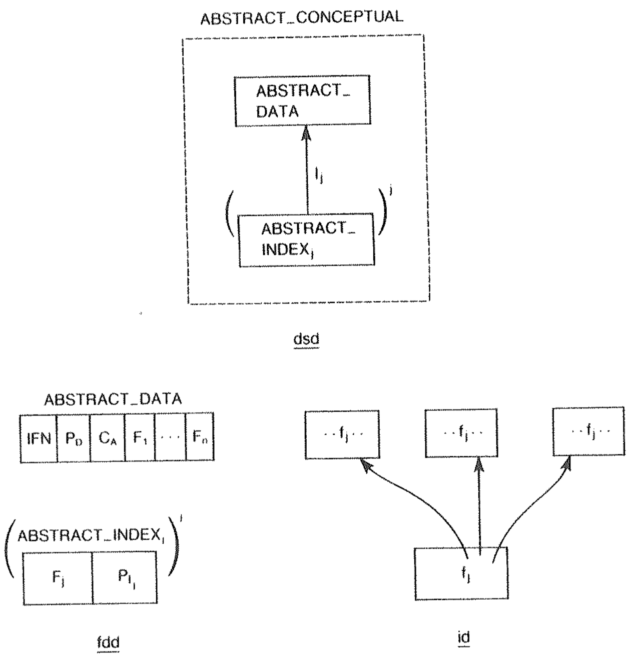


Fig. 52. Extraction of ABSTRACT_CONCEPTUAL data fields.

The data fields $F_1 \ldots F_n$ and inverted list fields P_{I_j} of ABSTRACT_INDEX$_j$ and ABSTRACT_DATA records are mapped to their internal counterparts by a conditional application of elementary transformations. The conditions under which a transformation is applied depends on the length of the data value and the length of the field in which it is to be stored. These mappings and their transformation models are explained in the following paragraphs.

When a CONCEPTUAL record type is defined, each field F_j is given a nominal length len$_j$. If the length of a data value to be stored in F_j is shorter than len$_j$ bytes, the data value is stored left-justified with blank padding. If it is longer, the first len$_j$ − b bytes are stored in the field and the remaining bytes are stored a single EFT$_j$ (extended field table) record. A pointer of length b bytes connects the "overflowed" field to the EFT$_j$ record. All data fields are represented in this manner.[21]

This materialization is modeled by two transformations: the *null* transformation describes the case where a data value has a length less than or equal to len$_j$ bytes. The *division* transformation captures the other case where a data value is divided into two fragments: the first fragment is stored with the original record, the second is stored as an EFT$_j$ record. Both records are connected by a singular pointer linkset. This materialization is referred to as the *overflow transformation.*

Each ABSTRACT_INDEX$_j$ record contains an inverted list field P_{I_j}. SYSTEM 2000 stores the contents of this field in one of two ways. If there is precisely one pointer in P_{I_j}, the field is not modified. (This is modeled by the *null* transformation.) If there are two or more pointers, P_{I_j} is *divided* into variable-length fragments called MOT$_j$ (multiple occurrences table$_j$) records. Link M_j, which connects the index record to its MOT records, is realized as a multilist with last child pointers. MOT$_j$ records are linked in chronological order. This materialization is referred to as the *inverted list transformation.*

An ABSTRACT_INDEX$_j$ record is materialized by applying the *overflow transformation* to field F_j and the *inverted list transformation* to field P_{I_j}. A DVT$_j$ (distinct value table$_j$) record is produced as a result. Also, link V_j and an EFT$_j$ records are produced if field F_j is divided, and link M_j and MOT$_j$ records are produced if field P_{I_j} is divided. Thus an ABSTRACT_INDEX file is mapped to one or more (internal) files in one of four different ways. Figure 53 illustrates each of these ways.[22] Note that the P_{M_j} and P_{I_j} fields in Figure 53.fdd occur in mutually exclusive situations and that both have the same length. Thus, for a given j, all records of the DVT$_j^{(1)}$ .. DVT$_j^{(4)}$ files share the same fixed length. This enables the records of all four DVT$_j^{(i)}$ types to be organized by a single file structure.

The ABSTRACT_DATA file of Figure 52 is materialized by segmenting the IFN, P_D, C_A fields from the data fields $F_1 \ldots F_n$. This produces an HT (hierarchical table) file and an ABSTRACT_DT file connected by link H. H is realized as a singular pointer (see Figure 54).

An ABSTRACT_DT record is materialized by applying the *overflow* transformation to each of its data fields. The resulting data record is referred to as a DT (data table) record; link E_j connects it with at most one EFT$_j$ record for each data field F_j. Figure 55.id illustrates a DT record with three data fields that have overflowed. Owing to the nature of the overflow transformation, DT records have a fixed length (which equals the sum of the nominal field lengths of the corresponding fields of the CONCEPTUAL record type). DT records of different CONCEPTUAL record types will, of course, have different lengths. EFT$_j$ records have variable lengths.

It is worth noting that if a data value overflows its nominal field length and that it occurs multiple times in a CONCEPTUAL file, there will be an EFT$_j$

[21] As numeric data values are fixed-length, division actually occurs only for nonnumeric data values. To distinguish between data values that are divided from those that are not, SYSTEM 2000 restricts the set of characters that can appear in a nonnumeric field. This enables a bit flag to be encoded within a character sequence to make the distinction.

[22] SYSTEM 2000 actually stores the pointer to the last MOT$_j$ record of an M_j link occurrence in the first MOT$_j$ record. A slightly more efficient implementation would store the last pointer in the DVT$_j$ as shown in Figure 53.id.

MOT records are variable-length. When a database is first loaded, all pointers of an inverted list are placed in a single MOT record. Subsequent pointer insertions are placed in new MOT records. The length of a new MOT record is a function of the length of the first MOT record.

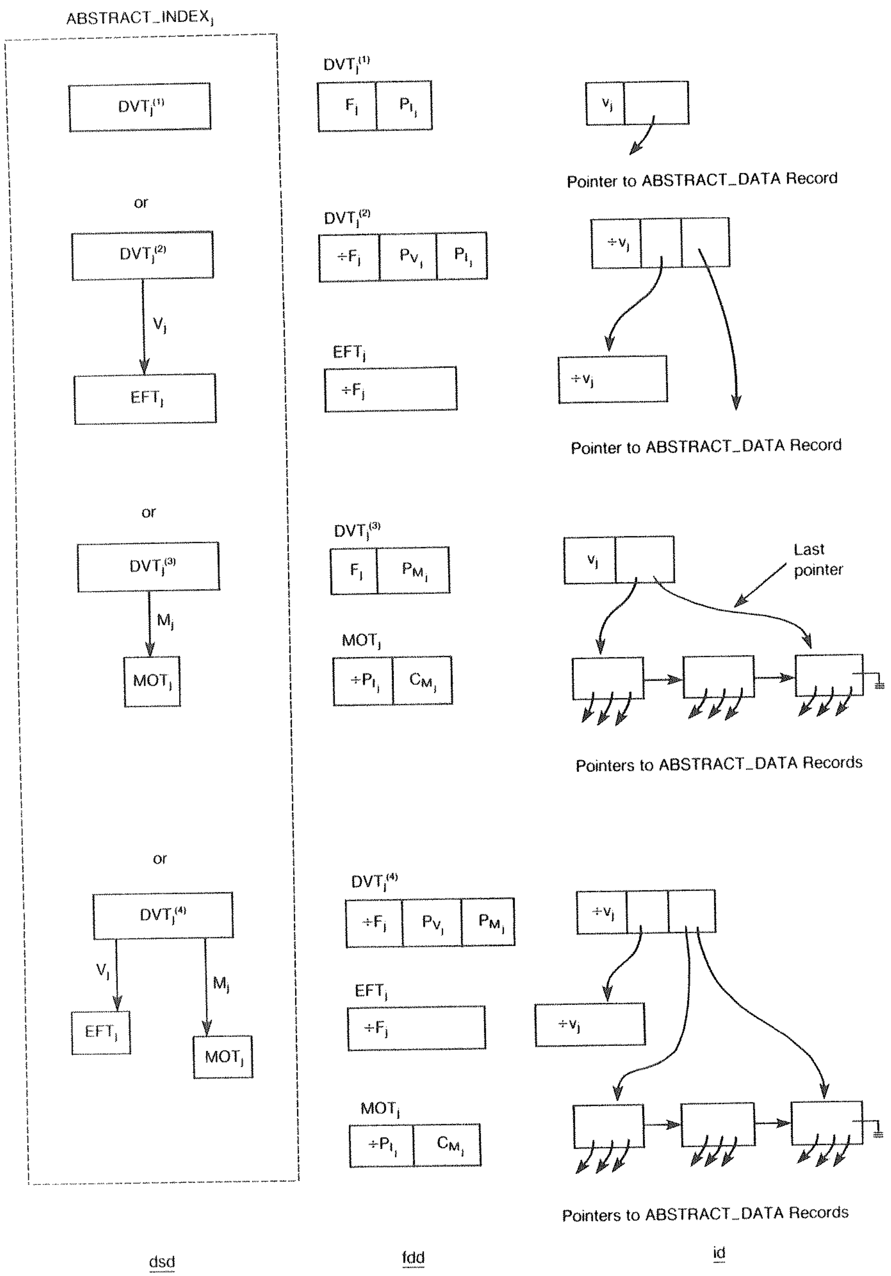


Fig. 53. Overflow and inverted list transformation of ABSTRACT_INDEX$_j$.

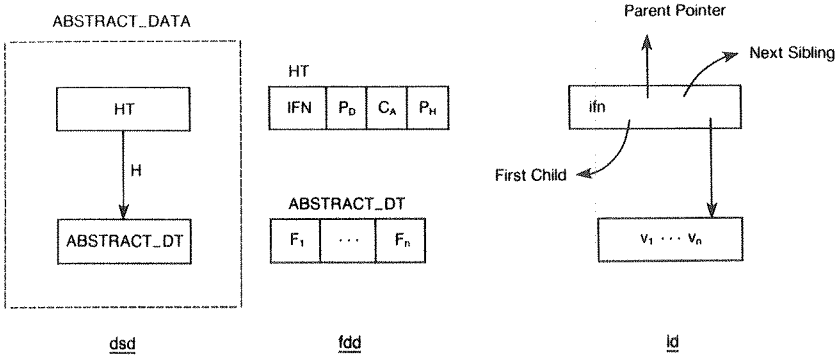


Fig. 54. Segmentation of ABSTRACT_DATA records.

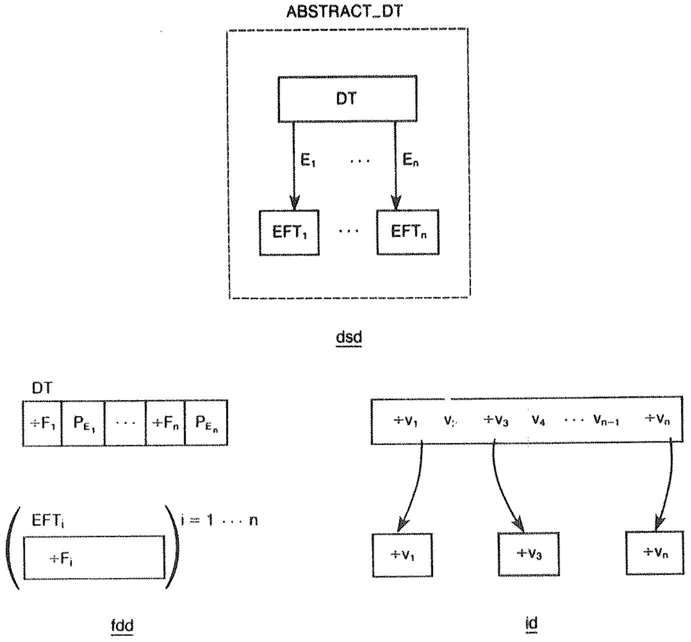
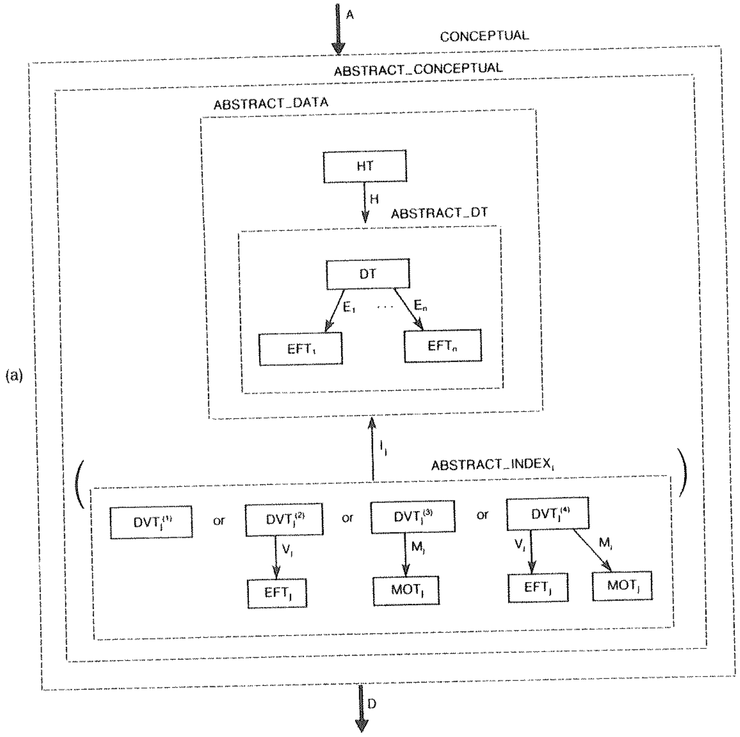


Fig. 55. Overflow transformation of fields of ABSTRACT_DT records.



Abstract File	Elementary Transformations
CONCEPTUAL	Augmentation and collection
ABSTRACT_CONCEPTUAL	Extraction
ABSTRACT_INDEX$_j$ (for all j)	Overflow and inverted list
ABSTRACT_DATA	Segmentation
ABSTRACT_DT	Overflow

(b)

Internal File	Simple File
HT	HT_SF
DT	DT_SF
EFT$_j$ (for all j)	EFT_SF
DVT$_j^{(i)}$ (for all i, j)	DVT_SF$_j$
MOT$_j$ (for all j)	MOT_SF

(c)

Simple File	Implementation
HT_SF	Unordered
DT_SF	Unordered
EFT_SF	Unordered
DVT_SF$_j$ (for all j)	B+ tree
MOT_SF	Unordered

(d)

Link	Linkset
H	Singular pointer
E$_i$ (for all i)	Singular pointer
V$_j$ (for all j)	Singular pointer
M$_j$ (for all j)	1 : N list with last pointer
I$_j$ (for all i)	1 : N inverted list
D	1 : N list with parent pointers
A	1 : N list with parent pointers (and last pointer if CONCEPTUAL file is root file of hierarchy)

(e)

Fig. 56. The storage architecture of SYSTEM 2000.

record for each of its occurrences if the data field itself is not indexed. SYSTEM 2000 eliminates duplicate EFT$_j$ records for data fields that are indexed.[23]

The internal files of SYSTEM 2000 are DVT$_j^{(i)}$, MOT$_j$, HT, DT, and EFT$_j$. For each j, files DVT$_j^{(1)}$ through DVT$_j^{(4)}$ are stored in a single B+ tree. (Thus, for each value of j, there will be a distinct B+ tree). All MOT$_j$ files are stored in a single unordered file. The HT and DT files are stored in separate unordered files, and all EFT$_j$ files are stored in a single unordered file.[24]

Figure 56 summarizes the storage architecture of SYSTEM 2000. Source materials are [16, 27, 50] and [80].

ACKNOWLEDGMENTS

I gratefully acknowledge the encouragement, support, and ideas I received from the following people: Ignacio Casas at the University of Toronto; Jim Desper at Infodata Systems, Inc.; John McCarthy at Lawrence Berkeley Laboratories; Alex Buchmann at IIMAS; Alan Wolfson at Software AG; Paul Butterworth at Relational Technology; Barbara Foster at Intel; Stan Su, Sham Navathe, and Jim Parkes at the University of Florida, and Tim Wise at the University of Texas. I thank the referees for suggesting several important improvements to the paper. I also thank Arie Shoshani (Lawrence Berkeley Laboratories) for his considerable assistance and insight in improving the clarity of this paper.

REFERENCES

1. AGHILI, H., AND SEVERANCE, D. G. A practical guide to the design of differential files for recovery of on-line databases. *ACM Trans. Database Syst. 7*, 4 (Dec. 1982), 540–565.
2. ALSBERG, P. A. Space and time savings through large database compression and dynamic restructuring. *Proc. IEEE 63*, 8 (Aug. 1975), 1114–1122.
3. BAROODY, A. J., AND DEWITT, D. J. An object-oriented approach to database system implementation. *ACM Trans. Database Syst. 6*, 4 (Dec. 1982), 576–601.
4. BATORY, D. S. On searching transposed files. *ACM Trans. Database Syst., 4*, 4 (Dec. 1979), 531–544.
5. BATORY, D. S. Optimal file designs and reorganization points. *ACM Trans. Database Syst. 7*, 1 (Mar. 1982), 60–81.
6. BATORY, D. S. Conceptual-to-internal mappings in commercial database systems. *ACM PODS 1984*, 70–78.
7. BATORY, D. S. Unpublished manuscript, 1984.
8. BATORY, D. S. Progress toward automating the development of database system software. *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. S. Batory, Eds., Springer Verlag, New York, 1985.
9. BATORY, D. S. GENESIS—An internal database compiler: Status report. In preparation.
10. BATORY, D. S., AND GOTLIEB, C. C. A unifying model of physical databases. *ACM Trans. Database Syst. 7*, 4 (Dec. 1982), 509–539.

[23] Note that our model does not capture this aspect of EFT records; we model all fields by duplicating EFT records. It is believed that this problem can be rectified by casting the TM in terms of an object-oriented model. This would allow the contents of a data field to be treated as an object. When a data field is duplicated (as in extraction with duplication), the object that it contains is not actually duplicated, but rather a reference to the object is created. When the object is stored, all references will point the single object instance.

[24] A SYSTEM 2000 database has another file called the DEFIN file. It contains metadata, such as the root nodes of all simple files and the system • record.

11. Batory, D. S., and Kim, W. Modeling concepts for VLSI CAD objects. *ACM Trans. Database Syst. 10*, 3 (Sept. 1985), 322–346.
12. Berelian, E., and Irani, K. B. Evaluation and optimization. In *Proceedings VLDB 1977*, 545–555.
13. Bohl, M. *Introduction to IBM Direct Access Storage Devices*. SRA, 1981.
14. Burnett, R. A. A self-describing data file structure for large data sets. In *Computer Science and Statistics: Proceedings of the 13th Symposium of the Interface*, Springer Verlag, New York, 1981, 359–362.
15. Casas, I. R. Analytic modeling of database systems: The design of a System 2000 performance predictor. M.Sc. thesis, Dept. of Computer Science, Univ. of Toronto, 1981.
16. Casas, I. R. Technical discussion, Univ. of Toronto, 1982.
17. Casas, I. R. Performance prediction of database systems. Ph.D. thesis, Dept. of Computer Science, Univ. of Toronto, 1985.
18. Ceri, S., and Pelagatti, G. *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.
19. Chen, P. P. S. The entity-relationship model—Toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar. 1976), 9–36.
20. Cincom Systems, Inc. *TOTAL PDP-11 Programmers Reference Manual*. Cincinnati, Ohio, 1979.
21. Comer, D. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June 1979), 121–138.
22. Cullinane Database Systems, Inc. *IDMS System Overview*, Westwood, Mass., 1981.
23. Das, K. S., and Teorey, T. J. Detailed specifications for the file design analyzer. Tech. Rep. 87, Systems Engineering Lab., Univ. of Michigan, Ann Arbor, 1975.
24. Date, C. J. *An Introduction to Database Systems*, 3rd ed., Addison-Wesley, Reading, Mass., 1982.
25. Desper, J. Technical discussion, Infodata Systems, Inc., 1983.
26. Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible hashing—A fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3 (Sept. 1979), 315–344.
27. Foster, B. Technical discussion, Intel Corp., 1984.
28. Fredkin, E. Trie memory. *Commun ACM 3* (1960), 490–500.
29. Fry, J. P., et al. Stored-data description and data translation: A model and language. *Inf. Syst. 2* (1977), 95–148.
30. Gambino, T. J., and Gerritsen, R. A database design decision support system. In *Proceedings of the ACM Conference on Very Large Data Bases* (1977), ACM, New York, 534–544.
31. Garza, J. F. Design and implementation of JUPITER: A general file management system. M.Sc. thesis, Dept. of Computer Science, Univ. of Texas at Austin, 1985.
32. Gesellshaft fuer Mathematik und Datenverarbeitung. *ADABAS: Database Systems Investigation Report*, vol. 2, part 1, Institute fuer Informationssysteme, Bonn, 1976.
33. Gotlieb, C. C. Some large questions about very large databases. In *Proceedings of the ACM Conference on Very Large Data Bases* (1981), ACM, New York, 3–7.
34. Gray, J. N. Practical problems in database management—A position paper. *ACM SIGMOD*, 1983, 3.
35. Guttag, J. Abstract data types and the development of data structures. *Commun. ACM 20*, 6 (June 1977), 396–404.
36. Guttman, A., and Stonebraker, M. Using a relational database management system for computer-aided design data. Electronics Research Laboratory, UCB/ERL M82/37, Univ. of California, Berkeley, 1982.
37. Hammer, M. Data abstractions for databases. In *Proceedings of the Conference on Data: Abstractions, Definitions, and Structure. SIGPLAN Not. 11* (1976), 58–59.
38. Haskin, R., and Lorie, R. On extending the functions of a relational database system. *ACM SIGMOD* (1982), 207–212.
39. Held, G. D., and Stonebraker, M. R. B-trees reexamined. *Commun. ACM 21*, 2 (Feb. 1979), 139–142.
40. Hoffer, J. A. A clustering approach to the generation of subfiles for the design of a computer database. Ph.D. dissertation, Cornell Univ., Ithaca, N.Y., 1975.
41. Hsiao, D., and Harary, F. A formal system for information retrieval from files. *Commun. ACM 13*, 2 (Feb. 1970), 67–73.

42. Hsiao, D. Ed. *Advanced Database Machine Architecture*. Prentice-Hall, Englewood Cliffs, N.J., 1983.
43. Huffman, D. A. A method for the construction of minimal redundancy codes. In *Proceedings IRE 40* (Sept. 1952), 1098–1101.
44. IBM Corp. *IMS/VS Version 1: Database Administration Guide*. San Jose, Calif., 1981.
45. Infodata Systems, Inc. *INQUIRE Basic Training Course*. Pittsford, N.Y., 1979.
46. Infodata Systems, Inc. *INQUIRE Database Design and Loading Manual*. Pittsford, N.Y. 1979.
47. King, R. P., Korth, H. F., and Willner, B. E. Design of a document filing and retrieval service. *ACM SIGMOD*, 1983. (Business and Office Databases).
48 Knuth, D. E. *The Art of Computer Programming, vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1983.
49. Kornatowski, J. Z. *The MRS User's Manual*. Computer Systems Research Group, Univ. of Toronto, 1979.
50. Kroenke, D. *Database Processing*, S.R.A. Inc., Chicago, 1977.
51. Larson, P. Dynamic hashing. *BIT 18* (1978), 184–201.
52. Lempel, A. Cryptology in transition. *ACM Comput. Surv. 11*, 4 (Dec. 1979), 285–305.
53. Lorie, R., et al. User interface and access techniques for engineering databases. To appear in *Query Processing in Database Systems*, Springer Verlag, New York, 1984 (see [9]).
54. March, S. T. Techniques for structuring database records. *ACM Comput. Surv. 15*, 1 (Mar. 1983), 45–80.
55. March, S. T., and Severance, D. G. The determination of efficient record segmentations and blocking factors for shared data files. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 279–296.
56. March, S. T., Severance, D. G., and Wilens, M. Frame memory: A storage architecture to support rapid design and implementation of efficient databases. *ACM Trans. Database Syst. 6*, 3 (Sept. 1981), 441–463.
57. Maxwell, W. L., and Severance, D. G. Comparison of alternatives in an information system. In *Proceedings Wharton Conference on Research on Computers in Organizations* (Oct. 1973), Univ. of Pennsylvania, Philadelphia, 1–16.
58. Mitoma, M. F., and Irani, K. B. Automatic database schema design and optimization. In *Proceedings of the ACM Conference on Very Large Data Bases* (1975), 286–321.
59. Nakamura, T., and Mizoguchi, T. An analysis of storage utilization factor in block split data structuring scheme. In *Proceedings of the ACM Conference on Very Large Data Bases* (1978), 489–495.
60. National Bureau of Standards *Federal Information Processing Standards*, Publ. 46, 1977.
61. NDX Retrieval Systems, Inc. *CREATABASE Performance Manual*. Houston, Tex., 1981.
62. Nievergelt, J., Hinterberger, H., and Sevcik, K. C. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst. 9*, 1 (Mar. 1984), 38–71.
63. Ong, J., Fogg, D., and Stonebraker, M. Implementation of data abstraction in the relational database system INGRES. *ACM SIGMOD Rec. 14*, 1 (Mar. 1984), 1–14.
64. Ritchie, D. M., and Thompson, F. The UNIX time-sharing system. *Commun. ACM 17*, 7 (July 1974), 365–375.
65. Rowe, L. A., and Shoens, K. A. Data abstractions, view, and updates in RIGEL. In *Proceedings ACM SIGMOD* (1979), 71–81.
66. Scholl, M. New file organizations based on dynamic hashing. *ACM Trans. Database Syst. 6*, 1 (Mar. 1981), 194–211.
67. Senko, M. E., Altman, E. B., Astrahan, M. M., and Fehder, P. L. Data structures and accessing in database systems. *IBM Syst. J. 12*, 1 (1973), 30–93.
68. Severance, D. G. Some generalized modeling structures for use in design of file organizations. Ph.D. thesis, Univ. of Michigan, Ann Arbor, 1972.
69. Severance, D. G., and Lohman, G. M. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 256–267.
70. Shipman, D. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 2*, 2 (June 1977), 105–133.
71. Software AG of North America, Inc. *ADABAS: Introduction*. Reston, Va., 1977.
72. Software AG of North America, Inc. *ADABAS: Effective Data Base Management for the Corporate Environment*. Reston, Va., 1980.

73. SPERRY-UNIVAC *DMS-1100 Schema Definition: Data Administrator Reference.* 1975.

74. STANFORD UNIV. *Design of SPIRES: Vols. I and II.* Center for Information Processing, Stanford Univ., 1973.

75. STATISTICS CANADA *RAPID Internals Manual.* Ottawa, Ont., 1981.

76. STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.

77. STONEBRAKER, M., RUBENSTEIN, B., AND GUTTMAN, A. Application of abstract data types and abstract indices to CAD data bases. In *Proceedings 1983 ACM Engineering Design Applications,* ACM, New York, 107–114.

78. SVENSSON, P. On search performance for conjunctive queries in compressed, fully transposed ordered files. In *Proceedings of the ACM Conference on Very Large Data Bases* (1979), 155–163.

79. TEOREY, T. J., AND FRY, J. P. *Design of Database Structures.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

80. TSICHRITZIS, D. C., AND LOCHOVSKY, F. *Data Base Management Systems.* Academic Press, New York, 1977.

81. TSICHRITZIS, D. C., AND KLUG, A., EDS. The ANSI/X3/SPARC DBMS framework report of the Study Group on Database Management Systems. *Inf. Syst. 3*, (1978), 173–191.

82. TSICHRITZIS, D., AND CHRISTODOULAKIS, S. Message files. *ACM Trans. Office Inf. Syst. 1*, 1 (Jan. 1983), 88–98.

83. TURNER, M. J., HAMMOND, R., AND COTTON, P. A DBMS for large statistical databases. In *Proceedings of the ACM Conference on Very Large Data Bases* (1979), 319–327.

84. WELLS, M. File compression using variable length encodings. *Comput. J. 15*, 4 (1972) 308–313.

85. WIEDERHOLD, G. *Database Design,* 2nd ed., McGraw-Hill, New York, 1983.

86. WISE, T. E. A technique to model and design physical database structures. M.Sc. thesis, Dept. of Computer and Information Sciences, Univ. of Florida, 1983.

87. WOLFSON, A. Technical discussion. Software AG of North America, Inc., 1983.

88. YAO, S. B. An attribute-based model for database cost analysis. *ACM Trans. Database Syst. 2*, 1 (Mar. 1977), 45–67.

89. ZIV, J. AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor. IT-23*, 3 (May 1977), 337–343.

Received June 1983; revised July 1984; accepted June 1985

Concepts for a Database System Compiler [*]

D.S. Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78711

Abstract

We propose a very simple formalism based on parameterized types and a rule-based algebra to explain the storage structures and algorithms of database management systems. Implementations of DBMSs are expressed as equations. If all functions referenced in the equations have been implemented, the software for a DBMS can be synthesized in minutes at little cost, in contrast to current methods where man-years of effort and hundreds of thousands of dollars are required. Our research aims to develop a DBMS counterpart to today's compiler-complier technologies.

1. Introduction

A fundamental problem in computer science is the search for a technology to construct customized software systems rapidly and cheaply. An important instance of this problem is in the area of database management. There are many engineering and scientific applications (e.g., AI, CAD, graphics) that need database support, yet current DBMSs cannot accommodate these applications easily or efficiently. The reason is that new data types, novel query processing algorithms, and specialized storage structures are among their requirements. Modifying existing DBMSs or building new DBMSs from scratch is an exceedingly costly and difficult undertaking. What is needed is a technology that can be used to construct customized DBMSs quickly and cheaply.

Intuitively, such a technology would have the following features. New algorithms would be encapsulated in standardized units (like today's hardware chips) which could be plugged into DBMSs. Different manufacturers would feature different lines of algorithms/units to enable users to upgrade or enhance their systems, much in the same way that hardware systems can be upgraded today. Technology transfer times would be measured in man-months, not man-decades. Database systems would no longer be monolithic, but synthetic.

In this paper, we explain how such a technology is possible and the ideas on which it rests. Customizing DBMSs is the domain of **extensible database systems**. The ideas that we present here underly the design of GENESIS [Bat82-88]. Other extensible database systems, such as EXODUS, STARBURST, POSTGRES, and DASDBS, are based on different ideas or have different goals than those of GENESIS. They are discussed in detail in [IEE87].

The primary distinctions between GENESIS and other extensible DBMS projects is the emphasis on reusable building blocks and DBMS compiler/generators. Most projects, such as POSTGRES and STARBURST, are concentrating on the development of a fixed DBMS architecture that can accommodate new features. The EXODUS and GENESIS projects, in contrast, are aiming at generalized architectures with DBMS compiler/generators as their goal.

EXODUS provides tools (e.g., the E programming language) and a few fixed components (e.g., object manager and query optimzer) to simplify the task of DBMS software construction. The design of interfaces, the semantics of modules, and how modules interact are left up to the Database Implementor (DBI). Module reusability hinges on the latter design decisions, and is not a promoted feature of EXODUS.

GENESIS is based on a well-defined conceptual framework in which reusable building-block modules of DBMSs can be identified. Interfaces, module semantics, and possible module interactions are standardized. An open-ended library is provided, along with a tool (the compiler) to express and construct target DBMSs as compositions of modules. The coding of modules and the fitting of algorithms to standardized interfaces are left up to the DBI. We believe that the EXODUS and GENESIS approaches are complimentary.

We begin our discussions by explaining the approach we are taking to develop the GENESIS DBMS compiler. We then formalize these ideas in terms of an algebra, and illustrate them with simple examples. Insights into future work and distinctions between our work and recent contributions, namely those of Freytag [Fre87] and Graefe and DeWitt [Gra87], are presented as conclusions.

2. The Approach

The fundamental problem in developing a building-blocks technology for DBMSs is knowing how to decompose DBMSs in a way in which identified pieces are demonstratably reusable. The solution requires a very general and very simple framework in which known algorithms, known structures, and operational systems can be cast and related.

We believe two concepts are fundamental to a building-blocks framework/technology: standardized interfaces and layered DBMSs. How interfaces and layers are defined is the key to reusability. In this section, we explain our approach to their definition and overview their formalization.

2.1 Simplest Common Interface

A building-blocks technology requires an open architecture with standardized interfaces. The key to its extensibility lies in how these interfaces are designed. It is our belief that declaring an ad hoc interface to be a standard is the worst of all possibilities. A better approach is to 1) identify a class of algorithms that realize the same function, and 2) design the simplest interface that supports *all* algorithms of the class. The greater the number of algorithms, the more likely it is that *the interface captures fundamental properties of the algorithm class.* Such an interface is no longer ad hoc, but is justified by its demonstratable generality. We call this the **simplest common interface** (SCI) method for standardized interface design.

[*] This work was supported by the National Science Foundation under grant DCR-86-00738.

The SCI method promotes algorithm interchangability. All algorithms of a class support the same interface. If a particular algorithm doesn't provide the desired performance, it can be replaced by another without altering higher-level modules.

As an example, shadowing, page logging, and db-cache are three well-known database recovery algorithms. If one were to give each algorithm to a different implementor, three disparate interfaces would be designed. Algorithm interchangability would not be present. However, by defining an interface for all three, interchangeability is guarenteed.

The ingenuity of our colleagues ensures us that no single interface can always encompass all future algorithms. (Note this is also true for 'extensible' DBMSs with ad hoc interfaces). *However, a building-blocks technology can exploit the maturity of a field because ideas and concepts (e.g. interfaces) do reach a steady state.* An SCI exploits known algorithms. If an algorithm class is sufficiently large to begin with, adding a new algorithm will normally require no changes to its SCI. This reflects the common goal of researchers to develop *new* methods for solving *recognized* problems.

In our experience, a vast majority of SCIs are static. Look at the research in join algorithms over the last decade; the join operation hasn't changed, but there are lots of join algorithms that have been proposed. Similarly, a sort operation takes a stream of records and a sort criterion as input. It is unlikely that a sort algorithm will be invented that cannot be cast in terms of this interface. Of course, there are operations whose generalizations are not well understood. (We will consider some in Section 4.1). This means that new algorithms will occasionally be proposed that require evolutionary changes to an SCI. Radical modifications, which should be expected for ad hoc interfaces, are less likely.

SCI is a necessary, but not sufficient, requirement for a DBMS building-blocks technology. As an example, one could build a monolithic file management system that provides an SCI interface to all file structures. While the interchangeability of different structures is an important and recognized goal in DBMSs, there are lower-level primitives on which all file structures rely; the implementations of these primitives should not be duplicated. A better approach is to use a layered architecture, where each layer provides the primitives on which the next higher layer is defined. To provide maximum extensibility, each layer should have an SCI. How to slice DBMSs into layers brings us to the second concept.

2.2 Layered DBMSs and Conceptual-to-Internal Mappings

Database systems implement conceptual-to-internal mappings. We have shown in earlier papers [Bat84-85] that the data mappings of operational DBMSs can be very complicated, but have simple descriptions as a composition of primitive mappings. Storage structures are the end product of these mappings. An important contribution of our earlier work is that complicated storage structures are compositions of primitive structures.

Operation mappings accompany every primitive data mapping. The encapsulation of primitive data and operation mappings is a **layer**. Among known layers are indexing, compression, transposition, and horizontal partitioning. DBMSs are compositions of layers. Some layers are plug-compatible, so the order in which they can be combined is permutable. Other layers can be composed only in a certain order. In [Bat86a], we show how a DBMS was implemented by composing independently defined layers.

2.3 Overview of Formalization

This paper presents a formalization of the above ideas. It is this formalism that we are implementing in GENESIS. We show that the building blocks of DBMSs are **data types** (data + operations) and **algorithms** (implementations of operations). Each data type corresponds to a layer in the above discussions, and the interfaces to its operations are SCIs. Algorithms realize the operation mappings of these types. Reusability is a result of this formalization: each data type and its algorithms are defined independently of any DBMS in which it will be used. For this reason, it can be combined with other data types to form many DBMSs.

The fundamental data types of DBMSs fall into three classes: FILE, LINK, and MODEL. FILE is the class of data types that correspond to file implementations (e.g., B+ trees, inverted files, etc.). LINK is the class of data types that correspond to implementations of relationships among files (e.g., join indices [Val87], CODASYL sets). MODEL is the class of data models and their corresponding data languages.

Data types that are parameterized (e.g., STACK_OF[x]) can be composed with other types (e.g., INT) to form more complicated types (e.g., STACK_OF[INT]). Many of the types within the FILE, LINK, and MODEL classes are parameterized, and combinations of them can be identified with the architectures of recognized DBMSs. Section 3 explains this relationship in detail.

Associated with each class is a small set of generic operations that all members support. It is always possible for every member of the FILE class, for example, to perform record retrieval and record insertions. By cataloging algorithms that implement the generic operations of each of type in the FILE, LINK, and MODEL classes, it is possible to codify DBMS implementation knowledge as **rewrite rules** and to express and manipulate DBMS designs as **equations**. A simple algebra that accomplishes this is presented in Section 4.

Computations in centralized DBMSs are unified with the distributed and parallel computations of database machines and distributed DBMSs in Section 5. The unification is captured in the algebra by special rewrite rules and execution site assignments for algorithms.

We explain in Section 6 how the above ideas form a cohesive nucleus for a **database system compiler**. We conclude with Section 7 by pointing out the strengths and weaknesses of our approach, and how our work extends existing research.

3. Database Systems and Parameterized Types

Generic or **parameterized types** were proposed many years ago as a way to simplify software development and to promote software reusability [Lis77, Gog84]. A classical example of a parameterized type is STACK_OF[x]. One can define and implement stacks and stack operations independently of the objects that are placed on a stack. Parameterizing the STACK_OF type and the **module/layer** that implements the STACK_OF type attains this independence. So if POLYGON is a type, STACK_OF[POLYGON] defines a type that is a stack of polygons. The implementation of the composite type is a composition of the STACK_OF and POLYGON modules. Because both types and their modules were developed independently, they can be recognized as building blocks of more complicated systems.

Implementations of database management systems can also be viewed as a composition of types. Consider the nonparameterized types BPLUS, ISAM, and HEAP. Instances of these types are specific B+ tree, isam, and heap file structures.

Let FILE denote the class of all file implementations, of which the BPLUS, ISAM, and HEAP types are members. To promote uniformity and module interchangability, all members of the FILE class support *exactly* the same interface. This is possible, as one can always retrieve records, insert records, delete records, etc. from a file regardless of how it is implemented. By imposing a standardized interface, a program that references a file F whose type was BPLUS will still work if F's implementation is changed to HEAP. [1] (Providing, of course, that a type conversion - e.g., file unload and reload - takes place).

The FILE class also has parameterized types. Let INDEX[df:FILE, xf:FILE] be the type that specifies a parameterized implementation of an inverted file. The notation df:FILE means that parameter df can be assigned one or more types belonging to the FILE class. When a file is of type INDEX, the INDEX module/layer maps the file (henceforth called an **abstract file**) to a data file and zero or more index files (henceforth called **concrete files**). The key idea behind the parameterization is that the data and operation mappings of INDEX *do not rely* on the implementations of the concrete data file and concrete index files. For this reason, the file types of the data and index files are parameters to INDEX.

A common implementation of inverted files has data files implemented as heaps and index files as B+ trees. This corresponds to the type

[1] The imposition of a standardized interface is evident conceptually, but it is VERY rare to find DBMSs implemented in this way. The author is not aware of an exception where the interface to conceptual files matches that of internal files.

expression INDEX[HEAP,BPLUS]. Assigning different file structures to the data file and index files yields different inverted file implementations.

Each type encapsulates the data and operation mappings from an abstract file to one or more concrete files. As a few examples, ENCODE[ef:FILE] maps an unencoded/uncompressed file to an encoded/compressed file whose implementation is ef; XPOSE[sf:FILE] maps an untransposed file to a transposed file, where sf is the subfile implementation; and HPART[pf:FILE] horizontally partitions an abstract file, where pf is the implementation of a concrete file partition. The class of FILE implementations is quite large, having hundreds of members. A comprehensive list of known file structures (nonparameterized types) and file mappings (parameterized types) is given in [Bat84-85].

There are two additional building block classes besides FILE that underly DBMS implementations. One is the LINK class. Databases traditionally have been described as networks of files and links, where a link is a relationship between two files. LINK is the class of link implementations. Members of LINK include relational implementations, such as JALG (join algorithms) and JINDEX (join indices [Val87]), and CODASYL set implementations, such as PARRAY (pointer array) and RLIST (ring list). [2]

MODEL is the other class. A network interface to databases is archaic by today's standards. DBMSs therefore provide data models and data languages as their user interfaces. A MODEL type encapsulates the data and operation mappings that occur between a DBMS's user interface and its representation of databases as networks. QUEL[f:FILE, n:LINK] is a member of the MODEL class. It represents the data model and data language of INGRES, and is parameterized by f and n, which are respectively the implementations of the underlying files and links of INGRES databases [Sto76]. Other members of MODEL are SQL[f:FILE, n:LINK], the data model and data language of System R [Ast76], and DBTG[f:FILE, n:LINK], the primitive data model and data language of CODASYL [Dat82].

Four comments. First, just as STACK_OF and POLYGON can be recognized as building blocks of complicated systems, so too can the instances of FILE, LINK, and MODEL classes be recognized as building blocks of database management systems. INGRES, for example, corresponds to the composition:

QUEL[INDEX[dfile, ifile], JALG]

where dfile = { HEAP, ISAM, HASH, COMP_ISAM, COMP_HASH }

and ifile = { HEAP, ISAM, HASH, COMP_ISAM, COMP_HASH }

That is, INGRES presents a QUEL front-end, maps relations to inverted files, implements links by join algorithms, and allows data files and index files to be stored in one of five structures: heaps, isam, hash, compressed isam, and compressed hash. The latter two apply compression techniques as part of their file structure algorithms. (Each data page of a file structure is compressed. Note that the type compositions ENCODE[ISAM] and ENCODE[HASH] correspond to the situation where files are compressed *prior* to their storage in ISAM and HASH structures. ENCODE[ISAM] and ENCODE[HASH] are *not* the same as COMP_ISAM and COMP_HASH). The actual file structure that is used for a given data file or index file is specified by the physical database directive MODIFY-TO [Sto76].

As another example, RAPID, a statistical DBMS built by Statistics Canada in the mid-70's to process the Canadian census, has no data model front-end (i.e., RAPID has only a programming language interface), encodes input files, creates indices over selected encoded fields, stores index files in B+ trees, transposes the data file, and stores subfiles in heaps [Tur79]. This corresponds to the composition:

ENCODE[INDEX[XPOSE[HEAP], BPLUS]]

Taking other combinations of FILE, MODEL, and LINK modules yields other database systems. See [Bat84-85,87a] for additional examples.

Second, our approach to the description of DBMS implementations is **extensible**. New file implementations, link implementations, and data model/data language implementations are constantly being invented. They are accommodated easily as new FILE, LINK, and MODEL types.

Third, the ANSI/SPARC notions of a conceptual level and an internal level are present in our composite types [Dat82]. For every type expression, the most abstract files are **conceptual** and the most concrete files are **internal**. In the case of INGRES, relations correspond to the conceptual files (i.e., the abstract files of QUEL) and the files stored in HEAP, ISAM, HASH, COMP_ISAM, and COMP_HASH structures are internal. For RAPID, unencoded files are conceptual and the files stored in HEAP and BPLUS structures are internal.

Fourth, we have deliberately simplified our description of FILE, LINK, and MODEL types to provide a clean overview of our approach. It is not the case that standard notions of parameterized types suffices for our research; something much more sophisticated appears to be needed. Briefly, it is difficult to imagine how the interface to, say, INGRES could be defined in a programming language as the interface to a single data type (which we have called QUEL[f:FILE, n:LINK]). An instance of QUEL[] would be a database consisting of multiple relations. On a more mundane level, DBMSs create record types dynamically, and thus require dynamic type checking and the ability to define functions that are type-valued (i.e., functions that return data types as their result). We are not aware of any existing or proposed programming language that supports even this latter feature.

4. Rule-Based Algebras and Type Implementations

A layer is an implementation of a data type. We use a rule-based algebra to specify the implementation of layers and to show how algorithms compose when data types are composed.

Algorithms of FILE, LINK, and MODEL types are specified as a composition of functions. Functions represent either **algorithms** or **operations**, where the distinction is that algorithms implement operations. Generally, there are many algorithms for a given operation. For example, the sort operation can be implemented by a bubble sort algorithm, a quicksort algorithm, a radix sort algorithm, etc. We express the relationship between an operation O and the class of algorithms A_1, A_2, $\cdots$ that implement O as algebraic identities or **catalogs** of the form:

O	=>	A_1	; algorithm #1
		A_2	; algorithm #2
		...	

We distinguish two classes of algorithms: atomic and nonatomic. The most primitive algorithms, those whose decompositions are not considered interesting, are **atomic**. Compositions of atomic algorithms yield more complicated or **nonatomic** algorithms. Determining which algorithms are atomic is subjective; convenience is the best guide.

Catalogs provide us with two important capabilities: algorithm extensibility and algorithm interchangability. If a new algorithm A_n is invented for operation O, we simply add it to the catalog for O. A_n can then be used in the design and specification of a DBMS. This is **algorithm extensibility**. Similarly, if algorithm A_i does not exhibit desired performance characteristics, we simply replace it with another implementation of O. The DBMS will still work, but may run faster or slower. Swapping different implementations of operations is **algorithm interchangability**.

If an algorithm can process all instances of its operation, then it is **robust**. All popular sorting algorithms are robust. Quite often, certain instances of operations occur with such frequency that special algorithms are designed to handle them. For example, one can imagine a sorting algorithm that takes advantage of a preordering of its input elements. Because it works only under special conditions, and hence cannot process all instances of a sort operation, it is **nonrobust**. In general, one can always use robust algorithms as implementations for an operation. Using a nonrobust algorithm is legal only if restrictions are met.

The interface standardization of both the FILE and LINK classes requires that a small and fixed set of operations be supported by each class. For FILEs, one can retrieve, insert, delete, etc. records. For LINKs, one can

[2] Links are typically generated in the abstract-to-concrete mappings of files [Bat85]. As a rule, link implementations are implicit parameters to many FILE types. The inverted files of INDEX, for example, use links to connect index files with data files. To keep things simple, LINK parameters will remain implicit unless otherwise specified.

traverse links (i.e., compute joins), connect, and disconnect records. In contrast, MODEL types do not have standardized interfaces. This means that each MODEL type supports a different but fixed set of operations. Another source of operations are those that are performed on record streams. Streams of records arise as the result of computations; one can sort, filter, merge, split, etc. streams. The number of operations on streams is unrestricted. We explain in Section 4.3 how new operations on FILEs and LINKs, beyond those in the fixed set, are admitted.

In the previous section, we introduced type expressions as compositions of FILE, LINK, and MODEL types. In the following sections, we introduce a different class of expressions, called **algorithm expressions**, which are compositions of atomic algorithms. We illustrate algorithm expressions and the above ideas by cataloging implementations of operations in selected MODEL and FILE types. More complete catalogs are given in [Bat87a].

4.1 MODEL Implementations

Recall that MODEL is the class of types that represent different DBMS data models and data languages. Each MODEL type accomplishes the data mappings of user-defined objects and relationships to conceptual files and conceptual links, as well as the operation mappings from data language expressions to sequences of operations on these files and links.

In general, data mappings are simple. In relational DBMSs, for example, relations are mapped directly to conceptual files. Conceptual links, which are not explicit in relational models, map to join operations. The mapping of data language expressions to operations on conceptual files and conceptual links is much more complicated. Consider the subclass of MODULE types that implement relational data models and data languages.

Let R be a simple relational query that does not involve aggregations or the equivalent of nested SELECTs. (As readers will soon see, the particular relational language used to express R doesn't matter). The most abstract description of query processing in a relational DBMS is captured by the following expression:

```
EVAL( Q_OPT( R ) )
```

Q_OPT is the **query optimization** operation which maps R to an executable expression E. EVAL(E) executes expression E. Different relational DBMSs implement Q_OPT in different ways. To catalog recognized implementations, we note that Q_OPT is a well-known composition of three lower-level operations:

```
Q_OPT( R )   =>                                           (1)
     JOINING_PHASE( REDUCING_PHASE( Q_GRAPH( R ) ) )
```

Q_GRAPH:R→G maps a relational query R to a query graph [Ber81a, Yu84a-b], REDUCING_PHASE:G→G maps query graphs with unreduced files to graphs with reduced files, and JOINING_PHASE:G→E maps query graphs to executable expressions.

The Q_GRAPH operation has a large catalog of implementations; there is one implementation for each relational data language:

```
Q_GRAPH( R ) =>   SQL_GRAPH( R )         ; [Cha76]
                  QUEL_GRAPH( R )        ; [Sto76]
                  GEM_GRAPH( R )         ; [Zan83]
                      ...
```

The actual implementation of Q_GRAPH depends on the specific MODEL type (e.g., SQL_GRAPH(R) is used for the SQL type).

REDUCING_PHASE and JOINING_PHASE algorithms are data language and data model independent. A partial catalog of algorithms for each operation is shown below; more complete catalogs are given in [Bat87a]:

```
REDUCING_PHASE( G )    =>
    G                         ; identity
    SDD1( G )                 ; [Ber81b]
    BC( G )                   ; [Ber81a]
    YOL( G )                  ; [Yu84b]

JOINING_PHASE( G )     =>
    SYS_R( G )                ; [Sel79]
    U_INGRES( G )             ; [Won76]
    EXODUS( G, RS )           ; [Gra87] - RS is the rule set
```

It is important to note that the above algorithms *do not rely on specific implementations of conceptual operations*; rather, they are optimization strategies which enumerate or heuristically select orderings of operations on conceptual files and conceptual links. The details of how file operations are implemented is specified in the implementation of FILE modules, the subject of the next section.

A class of relational database query optimization algorithms follows from equation (1) as the cross product of the classes of algorithms that implement the Q_GRAPH, REDUCING_PHASE, and JOINING_PHASE operations. Specific compositions correspond to query algorithms of existing DBMSs. For example, the query optimization algorithm of SYSTEM R has the following definition:

```
Q_OPT(R)  =>  SYS_R( SQL_GRAPH( R ) )
```

That is, SYSTEM R uses 1) SQL_GRAPH to map SQL queries to query graphs, 2) the identity function G for its reducing phase algorithm, and 3) the SYS_R algorithm [Sel79] to map query graphs to executable expressions.

In principle, there are many other Q_OPT implementations that might have been used for SYSTEM R. For example, a DBMS with a SQL front-end that uses the University INGRES joining phase algorithm and the SDD1 reducing phase algorithm is:

```
Q_OPT(R)  =>  U_INGRES( SDD1( SQL_GRAPH( R ) ) )
```

Another possibility would be to develop a DBMS with an SQL front-end that uses the rule-based EXODUS joining phase algorithm with the SDD1 reducing phase algorithm:

```
Q_OPT(R)  =>  EXODUS( SDD1( SQL_GRAPH( R ) ), RS )
```

Many other combinations are possible. It is worth noting that most reducing phase algorithms are nonrobust (i.e., only tree graphs are handled by some algorithms). As a rule, nonrobust algorithms can be paired with existing robust algorithms to form new robust algorithms. For example, a new reducing phase algorithm NEW_REDUCE(G) is a composition of the BC algorithm (which works only on tree graphs) and the identity algorithm (which works on any graph):

$$\text{NEW\_REDUCE(G)} \;\; \Rightarrow \;\; \begin{cases} \text{BC(G)} & \text{; if G is a tree} \\ \text{G} & \text{; otherwise} \end{cases}$$

In this manner, the class of Q_OPT algorithms is actually much larger than the cross product of the Q_GRAPH, REDUCING_PHASE, and JOINING_PHASE classes.

Before proceeding, it is important to understand the behind-the-scenes role of the SCI method in the above catalogs. Q_GRAPH, REDUCING_PHASE, and JOINING_PHASE algorithms communicate through a standard definition of query graphs. Two comments. First, it is possible that the most efficient data structure (internal representation) for a query graph will vary among REDUCING_PHASE and JOINING_PHASE algorithms. To circumvent this problem, we express all REDUCING_PHASE and JOINING_PHASE algorithms in a query-graph-data-structure independent manner. All algorithms are expressed in terms of a standard set of operations on query graphs. One can then select the underlying data structure for query graphs that is jointly optimal for the REDUCING_PHASE and JOINING_PHASE algorithm pair. (Alternatively, one can choose the less optimal strategy of forcing all REDUCING_PHASE and JOINING_PHASE algorithms to use exactly the same data structure for query graphs. In either case, plug-compatibility among REDUCING_PHASE and JOINING_PHASE algorithms is achieved). This approach is examined in detail in [Bat88].

Second, query graphs may not be the best means for interalgorithm communication, in general. For simple relational queries, they work well. Difficulties arise when queries contain aggregation operations (e.g., average) or unusual operators (e.g., random sampling [Olk86]) are introduced. In addition, new REDUCING_PHASE or JOINING_PHASE algorithms are occasionally proposed that require new features to be added to query graphs that are not part of the 'standardized' definition.

Instead of generalizing query graphs, a more universal representation of a query is an expression that is formed by a composition of FILE and LINK operations (e.g., joins, semijoins, and file retrievals). An alternative to equation (1) would be the function composition:

$$
\begin{aligned}
&\text{Q\_OPT(R)} \quad \Rightarrow \\
&\qquad \text{JOINING\_PHASE\_EXPR(} \\
&\qquad\qquad \text{REDUCING\_PHASE\_EXPR(} \\
&\qquad\qquad\qquad \text{Q\_EXPR(R)))}
\end{aligned} \tag{1'}
$$

Q_EXPR would map retrieval statements to unoptimized expressions, REDUCING_PHASE_EXPR would map expressions on unreduced files to expressions on reduced files, and JOINING_PHASE_EXPR optimizes its input expression. (We note that exactly this approach - that of using expressions rather than query graphs - is being used in the EXODUS optimizer [Gra87]). The moral of this example is that finding the 'best' generalization may be a difficult research problem; but finding temporary/provisional solutions is fairly easy.

4.2 FILE Implementations

In this Section, we catalog the implementation of retrieval and insertion operations in three FILE types: HC, BXPOSE, and ENCODE. Each of these types has been shown to be important in effectively supporting scientific database applications [Sho85]. We will reconsider these types later in Section 6.

The file and stream operations that we will examine are INS, RET, and SORT. Let INS(F,r) be the operation that inserts record r into file F. Let RET(F,Q,O) be the retrieval operation that generates the stream of records from file F in O order that satisfy predicate Q, and let SORT(S,O) be the stream operation that sorts record stream S into O order.

The HC type is the header-compression file structure of Eggers, et al. [Egg81]. Basically, it is a variant of a B+ tree which has been tailored to the storage of simple, single-field, records. Run-length encoding is built into the structure, so that sequences of identical-valued records can be stored efficiently.

HC maps an abstract file AF to a header compression file structure F. Let INS_HC(F,r) be algorithm that inserts record r into a header-compression file F. Let RET_HC(F,Q) be the header-compression retrieval algorithm. It retrieves records from file structure F that satisfy predicate Q. Records are returned in the order in which they are stored. To retrieve records in a different order requires a sort. Thus the abstract retrieve and insertion operations have a single catalog entry:

$$\text{RET(AF,Q,O)} \quad \Rightarrow \quad \text{SORT(RET\_HC(F,Q), O)} \tag{2}$$

$$\text{INS(AF,r)} \quad \Rightarrow \quad \text{INS\_HC(F,r)} \tag{3}$$

Note that equation (2) is a template for generating a class of algorithms to implement RET(AF,Q,O). By substituting a different algorithm for the SORT operation in (2) yields a different implementation of RET(AF,Q,O).

ENCODE[ef:FILE] maps an abstract (or uncompressed) file to an encoded (or compressed) concrete file. There are many possible ways to encode files. The one we will examine is that of Wong, et al. [Won86]. The idea is to replace data values with binary codes (e.g., data value 'g' is replaced by the 4-bit code '0110'). How binary codes are associated to data values depends on the encoding algorithm; typical algorithms are binary, k-of-n, unary, superimposed, and order preserving codes [Bat83, Won86].

Selection predicates for retrievals are also mapped by ENCODE. For example, if field A is mapped to encoded field EA, then the predicate A='a' is mapped to EA=code('a'). Mappings of more complicated predicates (e.g., inequalities) are defined differently for each encoding. These details are not relevant to our discussions, but are discussed in [Bat83, Won86]. For our purposes, we simply note that ENCODE maps selection predicate Q on abstract files to predicate Q' on concrete (encoded) files.

Two entries in the RET catalog for ENCODE are:

$$
\begin{aligned}
&\text{RET(AF,Q,O)} \quad \Rightarrow \\
&\qquad \text{SORT(DECODE(RET(F,Q}',{*})\text{), O)} && (4) \\
&\qquad \text{SORT(FILTER(DECODE(RET(F,true,}{*})\text{)), Q), O)} && (5)
\end{aligned}
$$

That is, to retrieve records from abstract file AF in O order that satisfy predicate Q, one can either retrieve all encoded records that satisfy the encoded query, decode the selected records, and sort them in O order (eqn. (4)), or) read the entire encoded file, decode each record and apply the unencoded selection predicate Q, and sort the qualified records in O order (eqn. (5)). Other catalog entries, such as sorting records prior to decoding [Bat83], could also be listed.

Let COMP(r) be the algorithm that outputs the compressed version of record r. Inserting an abstract record is mapped to an insertion of its compressed counterpart:

$$\text{INS(AF,r)} \quad \Rightarrow \quad \text{INS(F, COMP(r))} \tag{6}$$

BXPOSE[bf:FILE] maps an abstract file to a bit transposed file [Won86]. The idea of BXPOSE is to transpose fixed-length records into n column files, where each column record is one-bit wide. Bit transposition is an extreme case of transposition.

Let $F_1 \cdots F_n$ be the column files to which AF is mapped. Also, let $\underset{i}{\xi} E_i$ be a notation for the expression list $E_1, E_2, \cdots$, where E_i is an expression. One entry in the RET catalog for BXPOSE is:

$$
\begin{aligned}
&\text{RET(AF,Q,O)} \quad \Rightarrow \\
&\qquad \text{SORT(FILTER(GLUE(} \underset{i \in QC(Q)}{\xi} \text{RET(}F_i\text{,true,}{*}\text{)), Q), O)}
\end{aligned} \tag{7}
$$

The RET(F_i,true,*) operation retrieves all bit records from F_i in their stored order. The column files that are read is given by QC(Q). (These are the columns that are needed for query selection and attribute projection). GLUE is the function that concatenates corresponding bit records of its input streams and forms a stream of abstract record fragments. Fragment records are then filtered and sorted into the desired order. Once again, this is one algorithm for implementing RET(AF,Q,O). Others are possible. [3]

Let BITS(r, $\overset{n}{\underset{i=1}{\xi}} r_i$) be the algorithm which transposes a record r to n bit-records $r_1 \cdots r_n$. Inserting an abstract record is accomplished by transposing the record and inserting each of its bit-records:

$$\text{INS(AF,r)} \quad \Rightarrow \quad \text{BITS(r, } \overset{n}{\underset{i=1}{\xi}} r_i\text{);} \quad \overset{n}{\underset{i=1}{\xi}} \text{INS(}F_i, r_i\text{)} \tag{8}$$

[3] Storing records whose size is a single bit can be very inefficient. A variation of BXPOSE algorithms is to pack many bit records into a larger record, possibly hundreds of bytes long, which is easier to store. Thus, when a RET from column file occurs, a small number of large records are returned. A different GLUE algorithm would unpackage streams of these larger records and produce the desired record fragments ready for FILTER to evaluate. This variation, among many others, can be captured in our model.

Cataloging algorithms for other operations, such as record modification and deletion, and for other FILE, LINK, and MODEL types is accomplished in an analogous way.

4.3 New Operations

As mentioned earlier, GENESIS admits only a small and fixed number of operations on FILEs and LINKs. This is essential for the plug-compatibility of FILE and LINK types. In nontraditional database applications, special-purpose operations on files and links are needed. These seemingly contradictory positions are resolved by recognizing that an unrestricted number of operations can be introduced on record streams, and that our catalogs can be generalized to allow a composition of operations to appear to the left of the =>. Consider the following example.

Suppose the aggregation operation COUNT(S) is to be introduced; it counts the number of records in stream S. The catalog for COUNT contains the simple algorithm count_alg(S) which increments a counter and outputs the contents of the counter when an end-of-stream marker is reached:

COUNT(S) => count_alg(S)

One can invent an algorithm for, say, B+ trees which searches a B+ tree F and returns the number of records that satisfy a query Q. Call it bplus_count(F,Q). It would be cataloged for the BPLUS type as:

COUNT(RET(F,Q)) => bplus_count(F,Q)

For a different FILE type, say INDEX, another algorithm would find the answer by taking the intersection and union of inverted lists:

COUNT(RET(F,Q)) => inv_count(F,Q,...)

where ... are the operations that retrieve index records. For other FILE types, there are efficient algorithms that implement the composition COUNT(RET(F,Q)).

The advantage of *not* introducing a new FILE operation can be seen in the following. Suppose one wants to construct a DBMS that has an ISAM file rather than a B+ tree. Also, suppose there is no *single* algorithm in the GENESIS library/catalog that implements COUNT(RET(F,Q)) for ISAM files. In such a case, GENESIS will compose the count_alg with the isam_retrieval algorithm to implement COUNT(RET(F,Q)). The target DBMS will still work, but it won't run as fast as a tailored algorithm that implements the composition COUNT(RET(F,Q)). Thus, the advantage of staying with a fixed set of FILE operators rather than allowing an infinitely expandable set is that it is unreasonable to introduce algorithms for *every* FILE type each time a new operator is introduced. Only for certain FILE types will such algorithms arise or make sense.

In this manner, an unlimited number of operations and algorithms on files (and links) can be admitted without sacrificing the simplicity of our framework.

5. Distributed DBMSs and Database Machines

The algebra that we have outlined in the previous sections can be generalized to describe the computations in distributed database systems and database machines. To express distributed computations, we adorn each function of an expression with a superscript to indicate the site at which it is to be executed. For example, the expression $\mathrm{JOIN}^c(\ \mathrm{SEMIJOIN}^d(A,B,J1),C,J2)$ performs a semijoin of files A and B over join predicate J1, and this result is JOINed with file C on join predicate J2. The semijoin is executed at site d and its result is shipped to site c where the join is performed. The shipping of files to different sites is thus implicit in the notation. (For example, files A, B, and C may need to have been shipped in order for the join and semijoin to have been processed at their specific sites).

Assigning different processors to execute functions or compositions of functions is sufficient to describe the algorithms used in distributed DBMSs, but is insufficient for database machines. Database machines exploit parallelism by distributing the computations of one or more operations over several concurrently-executing processors. The basic design principles at work can be expressed in terms of stream rewrite rules. Figure 5.1a shows a function G which consumes stream A and produces stream G(A). A rewrite of this expression that introduces parallelism into G's computation is accomplished by splitting stream A into substreams $X_1 \cdots X_n$, performing function G on each substream, and assembling the results $G(X_1) \cdots G(X_n)$ to reconstruct G(A). As an example, let G be the SORT operation. Sorting stream A can be accomplished by splitting A into substreams, sorting each substream in parallel, and merging the substreams.

An algebraic expression of Figure 5.1 is:

$$G(A) \;=>\; \mathrm{SPLIT}(\ A,\ \mathop{\xi}\limits_{j=1}^{n} X_j\ );\ \mathrm{ASSEMBLE}(\ \mathop{\xi}\limits_{j=1}^{n} G(X_j)\ ) \qquad (9)$$

The $\mathop{\xi}\limits_{j=1}^{n} X_j$ term of SPLIT is a list generating notation for $X_1 \cdots X_n$; its purpose is to assign labels to each of the n substreams generated by the split. ASSEMBLE merges n different streams into a single stream.

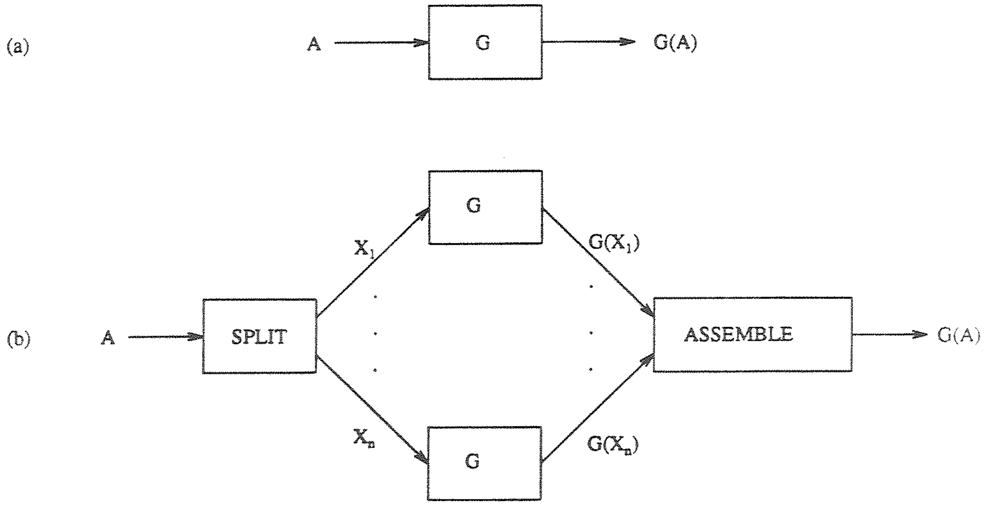


Figure 5.1 A Rewrite Rule for Database Machines

Rewrites such as (9) can be used to generate new algorithms from those given in existing catalogs. Consider equation (4), a catalog entry of the RET operation for ENCODE. Suppose it is observed that the DECODE algorithm takes so much CPU time that it is the performance bottleneck. To improve throughput, one could split DECODE's input stream into n substreams, DECODE n records in parallel, and merge the results. Applying rewrite (9) to equation (4) with G = DECODE, we obtain the desired parallel algorithm:

$$\Rightarrow \quad \text{SPLIT}(\,\text{RET}(F,Q',*)\,,\ \xi_{j=1}^{n} X_j\,);$$
$$\text{SORT}(\,\text{ASSEMBLE}(\,\xi_{j=1}^{n} \text{DECODE}(X_j)\,),\,O) \qquad (4')$$

With such rewrites, it is possible to specify various DBMS implementations for a parallel, multiprocessor setting. A catalog of rules that have been used in database machines and examples of retrieval algorithms used in a distributed DBMS and a database machine are given in [Bat87a].

6. Mechanics of a Database System Compiler

Our framework covers centralized DBMSs, distributed DBMSs, and database machines. In the following, we explain how a database system compiler would produce a centralized DBMS. Further research is needed to understand how distributed DBMSs and database machines can be 'compiled'.

An implementation of a centralized DBMS is specified in two steps. The first is to declare the architecture of the DBMS as a composition of MODEL, FILE, and LINK types. The second is to select the algorithms that implement each type's abstract-to-concrete operation mappings. Both steps are straightforward, and can be menu driven (e.g., select from a catalog of implementations). Once the specification is completed, a DBMS compiler composes the selected algorithms and simplifies using algebraic identities. The result are algorithm expressions that implement the target DBMS.

A classical example of a nontraditional database application is the storage and retrieval of scientific data [IEE84]. Scientific databases and their operations are quite different than those found in common business applications. Records typically have tens or hundreds of attributes, primarily containing numerical measurements or predefined codes (e.g., STATE has the domain {Alaska, Alabama, ..., Wyoming}). Retrieval operations generally reference all records of a file, but access only a few fields.

A customized DBMS for supporting such databases was proposed by Wong, et al. [Won86]. The architecture calls for the encoding of conceptual files so that the width of each field is reduced to a few bits. The encoded records are then mapped to a bit transposed file, where each subfile is stored in a header compression file structure. This architecture corresponds to the following composite type:

$$\text{ENCODE[BXPOSE[HC]]} \qquad (10)$$

No data model/data language was specified in [Won86].

The next step is selecting the algorithms to implement the abstract-to-concrete mappings of operations in each type. For the abstract retrieval operation, suppose we choose equation (4) for the ENCODE type, equation (7) for the BXPOSE type, and (2) for the HC type. For the insertion operation, equations (6), (8), and (3) are used.

At this point, the DBMS compiler takes over. The order in which algorithms are composed is specified by the composite type (10). That is, the concrete files of ENCODE are the abstract files of BXPOSE, and the concrete files of BXPOSE are the abstract files of HC. Note that the conceptual files of the target DBMS are the abstract files of ENCODE and the internal files are the concrete files of HC.

The algorithm to retrieve records from a conceptual file for this DBMS is obtained by composing equations (2), (4), and (7) as described above and eliminating unnecessary sorts. The latter is accomplished using the identity SORT(S,*) = S, i.e., sorting a sequence S in any order is identical to not sorting. The same holds for the insertion algorithm and the composition of equations (3), (6), and (8). Let O be an order. Let F, Q, and r denote a file, a query, and a record at the conceptual level, and let C_i, Q', and r_i be their internal counterparts (as prescribed in the mappings of (10)).

We obtain:

$$\text{RET}(F,Q,O) \quad \Rightarrow \quad \text{SORT}(\,\text{DECODE}(\,\text{FILTER}(\qquad (11)$$
$$\text{GLUE}(\,\xi_{i\in QC(Q)} \text{RET\_HC}(C_i,\text{true})\,),\ Q')),\,O)$$

$$\text{INS}(F,r) \quad \Rightarrow \quad \text{BITS}(\,\text{COMP}(r),\ \xi_{i=1}^{n} r_i\,);\ \ \xi_{i=1}^{n} \text{INS\_HC}(C_i,r_i) \qquad (12)$$

That is, retrieving conceptual records involves retrieving bit records from selected HC file structures, gluing corresponding bit records together, filtering the resultant record fragments using the encoded query, decoding the selected records, and sorting. Inserting a conceptual record involves compressing it, transposing it into bits, and inserting each of the bit records. The implementation of other conceptual operations, such as record modification and deletion, would be realized in a similar manner.

If all referenced algorithms are present in the library, it is a simple matter to link them together. (If they are not present, they will need to be coded and added to the library. Once coded, they can be reused). In this way, the construction of customized DBMSs can be rapid and inexpensive.

7. Contributions, Perspectives, and Current Limitations

In December 1985, we produced a working prototype which showed how a DBMS could be composed from independently defined layers. This work is described in [Bat86a]. The ideas on which our prototype was based were conceptually crude. This research arose from the need to formalize and refine these ideas in order to better understand and convey their capabilities and generality. Relating polymorphic types and rule-based algebras to our earlier work is a major step forward in our research. We are now in the process of upgrading the prototype to more clearly reflect the formalism we have presented in this paper.

Prior to and independent of our work, recent contributions have related rule-based algebras to query processing. The work of Graefe and DeWitt consider the problems of building efficient rule-based query optimizers [Gra87]. We are not concerned with implementing rule-based optimizers. Rather, our work is directed to the development of rule sets which may be used by such optimizers. We see our work complementing their research.

A second paper, by Freytag, presents a rule set for query optimization [Fre87]. His approach is to divide query optimization into phases, where each phase has its own rule set. Freytag's approach is quite similar to our own. However, there is a basic difference. *We are not dealing with query optimization. Rather, we are expressing DBMS implementations (e.g., retrieval and insertion algorithms) in terms of rule-based algebras.* Freytag's phases of query optimization are the mappings of retrieval operations through our layers (polymorphic types). Our research shows how Freytag's ideas can be understood in a much broader context. An important result of this generalization is a comprehensive framework to identify reusable building blocks (i.e., storage structures and algorithms) of DBMSs.

The immediate utility of our work is a formal means to specify the implementation of DBMSs. However, our goal is aimed at developing a DBMS compiler. There are still problems that need to be addressed in this framework; the more important ones are listed below. We believe all are solvable, and we indicate where the outlines of their solution can be found.

Concurrency Control. We describe DBMSs as compositions of layers. We need a theory of concurrency control for multi-layered software systems. Theories on multi-layered concurrency control already exist, but they presume that the software layers have already been identified. A marriage of our respective theories seems appropriate [Bat86a].

Performance Models. Performance models are needed by query optimizers to identify the most efficient strategy to process a query. Just as DBMS software can be constructed from standardized components, so too can performance models be constructed from components. The only difference is that one deals with atomic cost functions instead of atomic algorithms. [Bat87c] explains these ideas in more detail.

File Management Issues. The unparameterized FILE types in our framework, namely BPLUS, ISAM, etc., are implemented by the GENESIS file management system, named JUPITER. JUPITER is a composition of layers, much like our descriptions of DBMSs. However, JUPITER is composed of a fixed and preordered set of layers; among them are layers for buffer management, page-based recovery management, transaction management, and file management. A paper on JUPITER is forthcoming, and we aim to release JUPITER in early 1988.

New Data Types and Operators. Adding new data types and operators to the data model of a DBMS is an essential requirement of a DBMS compiler technology. We have developed a functional data model and data language as the basis for data type and operator extensibility in GENESIS. All computations in this model are expressed as compositions of functions, identical to the way in which we have specified the implementation of DBMSs in this paper. How these different aspects of our research relate is described in [Bat86b].

Hints to Lower Layers. Passing hints from upper layers to lower layers is often done to optimize performance. All of the examples of hint passing that are not already part of our model involves communications of higher level routines with the buffer manager. In general, passing hints is optional, and their inclusion is a DBMS design decision. We believe that there is nothing in our framework which precludes the addition of hint passing. We are currently studying examples to understand the general problem, and to see how it's solution fits into our framework.

8. Conclusions

Among the most important problems in computer science are software reusability and a means to construct customized software systems rapidly. Both problems are related. Software reusability hinges on recognizing building blocks of target systems, standardizing interfaces, and implementing modules independently of any system in which they can be used. Software systems can be constructed quickly by composing building blocks.

There are many examples of software reusability today. Database management systems are themselves reusable. DBMSs can be purchased 'off-the-shelf' and used in a variety of disparate applications. However, this granularity of reusability is too large to show how *different* DBMSs can be constructed from reusable components. It is this problem which we have addressed in this paper.

Our approach is to exploit the maturity of database research. We have presented simple methods for defining standardized interfaces/generic operations, and have reviewed how layered descriptions of DBMSs lead to the identification of reusable algorithms in DBMSs. We have formalized these ideas in terms of polymorphic types and rule-based algebras. Our work makes three potentially significant contributions. 1) It brings us a step closer to understanding the principles of how DBMSs can be constructed rapidly and cheaply from libraries of prewritten components, 2) it shows how DBMS designs can be formalized as an algebra, which is in contrast to the ad hoc ways system designs are expressed today, and 3) cataloging types and their algorithms is an important step in codifying and unifying knowledge of database system implementation.

Acknowledgements. I thank Arie Shoshani, Krzysztof Apt, and Jim Barnett for their helpful comments on an earlier draft of this paper.

References

[Ast76] M.M. Astrahan, et al., 'System R: 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.

[Bat83] D.S. Batory, 'Index Encoding: A Compression Technique for Large Statistical Databases', **Proc. Workshop on Statistical Database Management**, (1983), 306-314.

[Bat84] D.S. Batory, 'Conceptual-To-Internal Mappings in Commercial Database Systems', **ACM PODS 1984**, 70-78.

[Bat85] D.S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', **ACM Trans. Database Syst.**, 10,4 (Dec. 1985), 463-528.

[Bat86a] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K.Tsukuda, B.C. Twichell, and T.E. Wise, 'GENESIS: An Extensible Database Management System', to appear in **IEEE Trans. Software Engineering**.

[Bat86b] D.S. Batory, T.Y. Leung, and T.E. Wise, 'Implementation Concepts for an Extensible Data Model and Data Language', submitted. Also TR-86-24, Dept. Computer Sciences, University of Texas at Austin, 1986.

[Bat87a] D.S. Batory, 'A Molecular Database System Technology', submitted. Also TR-87-23, Dept. Computer Sciences, University of Texas, Austin, 1987.

[Bat87b] D.S. Batory, 'Principles of Database Management System Extensibility', in [IEE87], 40-46.

[Bat87c] D.S. Batory, 'Extensible Cost Models and Query Optimization in GENESIS', to appear in **IEEE Database Engineering**, 1987.

[Bat88] D.S. Batory, 'On the Reusability of Query Optimization Algorithms', to appear in **Information Sciences**, 1988.

[Ber81a] P.A. Bernstein and D.M. Chiu, 'Using Semi-Joins to Solve Relational Queries', **Jour. ACM**, 28,1 (Jan. 1981), 25-40.

[Ber81b] P.A. Bernstein, et al., 'Query Processing in a System for Distributed Databases (SDD-1)', **ACM Trans. Database Syst.**, 6,4 (Dec. 1981), 602-625.

[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.

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

[Egg81] S. Eggers, F. Olken, and A. Shoshani, 'A Compression Technique for Large Statistical Databases', **Proc. VLDB**, (1981), 424-434.

[Fre87] J.C. Freytag, 'A Rule-Based View of Query Optimization', **ACM SIGMOD 1987**, 173-180.

[Gra87] G. Graefe and D.J. DeWitt, 'The EXODUS Optimizer Generator', **ACM SIGMOD 1987**, 160-172.

[Gog84] J. Goguen, 'Parameterized Programming', **IEEE Trans. Software Engr.**, SE-10,5 (September 1984), 528-543.

[Gut77] J. Guttag, 'Abstract Data Types and the Development of Data Structures', **Comm. ACM**, 20,6 (June 1977), 396-404.

[IEE84] **Database Engineering**, Statistical Database Systems, 7,1 (March 84), D.S. Batory, ed.

[IEE87] **Database Engineering**, 10,2 (June 1987), M. Carey, ed.

[Lis77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, 'Abstraction Mechanisms in CLU', **Comm. ACM**, 20,8 (Aug. 1977), 564-576.

[Olk86] F. Olken and D. Rotem, 'Simple Random Sampling from Relational Databases', **VLDB 1986**, 160-169.

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

[Sho85] A. Shoshani and H.K.T. Wong, 'Statistical and Scientific Database Issues', **IEEE Trans. Software Engr.**, SE-11,10 (October 1985), 1040-1047.

[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.

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

[Val87] P. Valduriez, 'Join Indices', **ACM Trans. Database Syst.**, 12,2 (June 1987), 218-246.

[Won76] E. Wong and K. Youseffi, 'Decomposition - A Strategy for Query Processing', **ACM Trans. Database Syst.**, 1,3 (Sept. 1976), 233-241.

[Won86] H.K.T. Wong, J.Z. Li, F. Olken, D. Rotem, and L. Wong, 'Bit Transposition for Very Large Scientific and Statistical Databases', **Algorithmica**, 1 (1986), 289-309.

[Yu84a] C.T. Yu, Z.M. Ozsoyoglu, and K. Lam, 'Optimization of Tree Queries', **Jour. Comp. and Syst. Sci.**, 29,3 (Dec. 1984), 409-445.

[Yu84b] C.T. Yu and C.C. Chang, 'Distributed Query Processing', **Computing Surveys**, (Dec. 1984), 399-433.

[Zan83] C. Zaniolo, 'The Database Language GEM', **ACM SIGMOD 1983**, 207-218.

A Definition of Open Architecture Systems

with Reusable Components: Preliminary Draft

D.S. Batory and S.W. O'Malley
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

1. Introduction

The issues of domain modeling, defining open system architectures, and large scale software reuse converge when the goal is to create software systems from libraries of interchangeable components.

Genesis [Bat88-90], Avoca [Hut89, Oma90], and Choices [Cam90] are three independently conceived projects whose goal was to build complex software systems in the domains of database systems, communication networks, and operating systems by composing prefabricated components. All three projects exhibit strong similarities in terms of their conceptual design, organization, and implementation. These similarities are not accidental; we believe they are fundamental to interchangeable software component technologies.

In this paper, we outline a provisional model of open system architectures. The contribution of the model is a simple domain-independent formalism in which the results of a domain analysis can be expressed. As there is no currently accepted formalism, defining such models is an important research problem. What gives our particular model credibility is that it is based upon and unifies significant aspects of the designs of Genesis and Avoca (and possibly also Choices), which are among the largest known examples of domain modeling yet achieved. By modeling a domain in this formalism, one will have explicitly defined a software architecture for that domain which is inherently open, in addition to understanding how large-scale reuse within the domain can be achieved.

2. The Structure of Large Scale Software Systems

The structure of large scale software systems can be modeled by an elementary formalism that reflects the obvious fact that systems are designed as assemblies of components and that components fit together in very specific ways. The model postulates that components are instances of types and components themselves may be parameterized. The ways in which components fit together to form systems is captured elegantly through the use of typed parameters and typed expressions.

2.1 The Model Framework Framework and Notation

Components. The fundamental unit of software system construction is the *component*. Every component has an interface and an implementation. Following the lead of Parnas [Par79], the interface of a component is *anything* that is visible externally to the component. Everything else belongs to its implementation. In an object-oriented setting, a component may correspond to a single class or to clusters of closely-knit classes that act as a unit.

A fundamental concept of our model is that every component is an instance of some type AT, where all instances of AT present exactly the same interface, and each instance represents a different implementation. The interface of a component can be factored into type-specific and component-specific information. Type-specific information follows directly from an object-oriented design: it is the set of classes (their objects, operations, and interrelationships) that are visible. Component-specific information is descriptive, such as the components name, performance characteristics, source and object files, etc. Thus, when we say two components share the same interface or are plug-compatible, we are referring to the type-specific portion of their interface that they have in common.

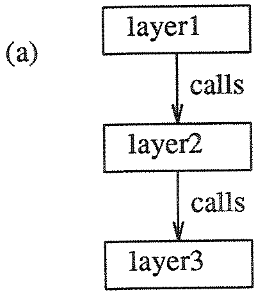
Enumerated Types and Parameterized Components. The set of all instances of a type AT is a *realm*. As a practical matter, most instances (components) of a realm are never implemented; few ever get beyond the paper-design stage. Those that are implemented define a *library*. We use an enumerated type notation:

AT = { a1, a2, a3 }

to mean that type AT has a1, a2, and a3 as library instances. Libraries are inherently *extensible*, since it is always possible to implement yet another instance of a realm.

Instances generally reference other components as parameters. Let the notation "t:T" mean that component t is of type T and "t:{ T }" means t is a set of one or more instances of T. Consider component c[x:R1, y:{ R2 }]. c has two parameters x and y, where x must be a component of type R1 and y must be a set of components of type R2.

A component can intuitively be thought of as a layer, where a software system is a stacking of different layers (i.e., composition of different components). It is normally the case that components can only be composed/stacked in a predefined order. Figure 1a shows the stacking of layers 1 - 3, where layer1 is on top and layer3 is on the bottom. (This stacking means layer1 calls layer2 for its lower-level services, and layer2 calls layer3 for its lower-level services. Layer3 is self-contained and requires services from no other component). Note that if the types for these layers (TOP, MIDDLE, and BOTTOM) are different, then only one composition of these layers is possible (Fig. 1b).



(b) TOP = { layer1[x : MIDDLE] }

MIDDLE = { layer2[y : BOTTOM] }

BOTTOM = { layer3 }

composition = layer1[layer2[layer3]]]

Figure 1. Nonpermutable Stacking of Layers

Reflexive Components. A distinctive and fundamental concept of our model is the possibility of *reflexive* components; components that can be composed (stacked) in arbitrary orders. More specifically, a component of type T is reflexive iff it has at least one parameter whose type is T. Components d[z:R] and e[z:R] of type R are reflexive as both have a parameter z which is of type R. Thus, compositions d[e[z:R]] and e[d[z:R]] are possible.

Unix filters are the prototypical examples of reflexive components. Piping the output of one filter to another is filter composition; because filters have the same interface, they can be composed in different orders. Usually, the order in which components are composed can make a substantial difference in performance and semantics.

UFilters = { dtbl[x:UFilter] , deqn[x:UFilter], ... }

Note that pipe expressions like dtbl | deqn correspond to deqn[dtbl] in our notation.

Without reflexive components, realms become exponentially larger than necessary. For example, distinct components representing every possible combination (e.g., de[z:R] for d[e[z:R]] and ed[z:R] for e[d[z:R]]) would have to be defined. Reflexive components are the "true" building-blocks of a realm; representing only their compositions avoids a significant opportunity for achieving component reuse on a very large scale.

Composition, Systems, and Domains. *Composition* is the rules and operations of component parameter instantiation; i.e., the guidelines by which components can be glued together. A *software system* is a composition of components that corresponds to a type expression where no component has an unbound parameter. The set of all software systems of type T is called the *domain* of T, denoted Domain(T).

A software tool that implements the rules of composition is a component *layout editor*; it provides a language in which type expressions can be written. The set of all systems of type T that can be specified by a layout editor from compositions of library components is called the *family* of T, denoted Family(T). Family(T) is always a subset of Domain(T). Figure 2 summarizes these concepts.

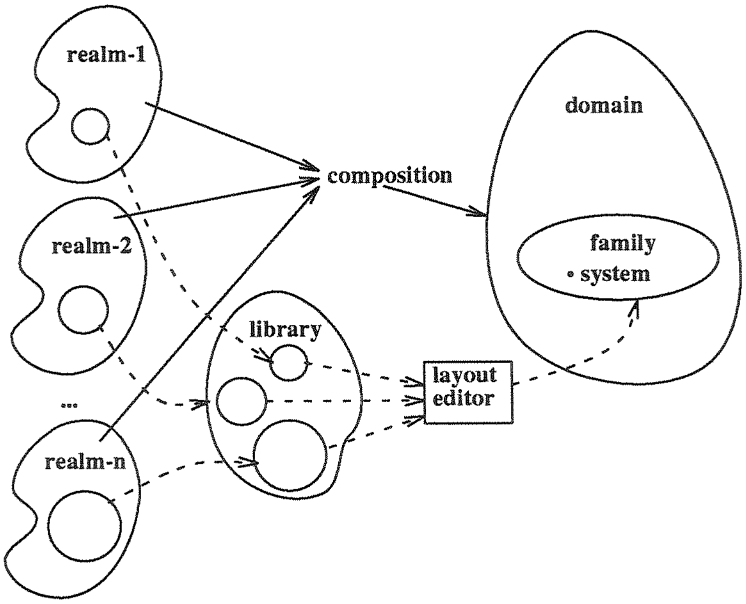


Figure 2. Realms, Libraries, Composition, Layout Editors, Families of Systems, and Domains

Component Reuse, Mega-Programming, and Open Architectures. Recognizing and achieving software reuse are fundamental problems in software engineering. An important form of reuse is *component reuse*, which in our model occurs when two or more expressions use the same component. Thus, if a[b[c]] and d[b[q]] are expressions (software systems), component b is being reused.

Mega-programming is the writing of type expressions (software systems). Each component is a fundamental unit of domain programming. The decomposition of a domain into realms defines an *open architecture* for that domain. It is open because each of its realms are extensible.

2. Examples from Genesis

A *storage system* is reponsible for mapping conceptual relations to disk. There are two realms from which storage systems are created: FMAP and AMETHOD.

FMAP is the realm of file mapping components. Each transforms a conceptual file to one or more internal files. What is "concrete" to one component, may be "abstract" to another, and hence most FMAP components are reflexive. Among the members of FMAP are components that accomplish secondary indexing (i.e., an abstract file is mapped to an inverted file), encoding (mapping an unencoded file to an encoded file), fragmentation (fragmenting long records into short records), and internal (mapping a relational-like file interface to a file structure/access method interface):

```
FMAP    =    { index[ d,i:FMAP ], encode[ d:FMAP ],
               frag[ s:FMAP ], internal[ d:AMETHOD ], ... }
```

AMETHOD is the realm of access methods. Among its members are nonkeyed access methods (heap, unord), single-keyed methods (hash, bplus, isam), and multi-keyed methods (grid).

```
AMETHOD    =    {  heap, unord, bplus, isam, hash, grid  }
```

A storage system is an expression of type FMAP. Storage system ss1, which maps conceptual relations to inverted files, where data files encoded prior to being stored in a heap structure, and index files stored in Bplus trees, is expressed by:

```
ss1  =  index[ encode[ internal[ heap ] ], internal[ bplus ] ]
```

Storage system ss2, which encodes long records, fragments them, and stores record fragments in unordered file structures, is expressed by:

```
ss2  =  frag[ encode[ internal[ unord ] ] ]
```

3. Examples from Avoca

An *asynchronous protocol suite* is a software system that sends and delivers messages asynchronously. There is a single realm from which asynchronous protocol suites are assembled: ASYNC.

ASYNC is the realm of primitive asynchronous protocols. Each protocol (component) transforms abstract messages to concrete messages, and vice versa. As a general rule, ASYNC protocols are reflexive. Among the members of ASYNC are AMD and Intel Ethernet drivers, the internet protocol (ip), user datagram protocol (udp), message fragmentation protocols (blast), and virtual protocols (vaddr - which routes messages according to whether the destination address is local or remote, and vsize - which routes messages according to the message size):

```
ASYNC    =    { amd_eth, intel_eth, ip[ x:ASYNC ], udp[ x:ASYNC ], blast[ x:ASYNC ],
                vaddr[ local,remote:ASYNC], vsize[ small,big:ASYNC ], ...}
```

An asynchronous protocol suite is an expression of type ASYNC. Suite (s1) is the standard implementation of the udp protocol suite: upd calls ip, which sits atop of an ethernet driver:

s1 = upd[ip[amd_eth]]

Common practice is to send messages through ip, even if messages are being delivered to machines on a local network. Suite (s2) eliminates this overhead by using vaddr to bypass ip and to place messages with local destinations directly on the ethernet:

s2 = upd[vaddr[amd_eth, ip[amd_eth]]]

4. Conclusions and Future Work

Developing formalisms to express the result of a domain analysis is an important research problem. The formalism we have presented is not only intuitively appealing and domain-independent, it also unifies important concepts in Unix, Genesis, and Avoca, three different systems that have realized large scale reuse. Although our model is provisional, we believe there are many domains that can be expressed in its formalism. If this is the case, a standard representation for domain analysis is within reach, and fundamental concepts underlying large scale reuse will have been identified.

5. References

[Bat88] D.S. Batory, "Concepts for a Database System Compiler", *ACM PODS* 1988.

[Bat90] D.S. Batory, "The Genesis Database System Compiler: User Manual", U. of Texas Tech. Rep. TR-90-27, 1990.

[Cam90] R.H. Campbell, "Considerations of Persistence and Security in Choices, an Object-Oriented Operating System", U. of Bremen Tech. Rep. 1990.

[Hut89] N.C. Hutchinson, S. Mishra, L.L. Peterson, and V.T. Thomas, "Tools for implementing Network Protocols", *Software-Practice and Experience*, Sept. 1989.

[OMa90a] S.W. O'Malley and L.L. Peterson, "A New Methodology for Designing Network Software", submitted for publication.

[Par79] D.L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE TOSE*, March 1979.

Construction of File Management Systems From Software Components *

D.S. Batory, J.R. Barnett, J. Roy, B.C. Twichell, and J. Garza

Department of Computer Sciences
The University of Texas
Austin, Texas 78711

Abstract

Domain analysis is a classical approach in software engineering to the identification of reusable software modules. It relies on indepth studies of existing systems, published algorithms and structures to descern generic architectures for large classes of systems. An architecture is a template in which building-block modules can be plugged. Interfaces are standardized to make blocks interchangable.

In this paper, we explain how domain analysis has lead us to a building-blocks technology for file management systems (FMSs) and we describe our most recent prototype, an FMS synthesizer. The synthesizer enables a customized FMS to be assembled from prewritten components in minutes at virtually no cost. Producing a comparable FMS from scratch would require man-years of effort and hundreds of thousands of dollars.

Keywords: domain analysis, software building blocks, software reusability, database systems

1. Introduction

It is well-known that existing relational database systems are inadequate for managing databases of CASE, VLSI CAD, and AI applications. Rather than building application-specific DBMSs from scratch, which is a cost-ineffective means of customization, current research has been directed to the development of extensible database systems, i.e., DBMSs that can be easily customized. Extensibility research embraces advances in both databases and software engineering.

Different notions of extensibility are being explored. POSTGRES uses a fixed architecture that accommodates a wide range of new features [Sto86]. EXODUS, in contrast, provides an open architecture with a few fixed components (a rule-based query optimizer and a storage manager) and tools (e.g., the E programming language) to simplify the burden of writing specialized DBMS software [Car86]. STARBURST stresses a building-blocks approach to query processing [Lee88]. Rule-sets more elaborate than those of EXODUS are used to identify and relate primitive query processing algorithms, and to specify legal transformations of access path expressions. GENESIS, our project, also relies on building-blocks. We differ from STARBURST in that we consider building-blocks for all operations - update, insertion, deletion, etc. - not just retrievals [Bat88a-b].

The goal of GENESIS is to demonstrate that customized DBMSs can be synthesized quicky from prewritten components. Enormous increases in software productivity are achieved by exploiting reusable and plug-compatible modules. The popularized - but mythical - concept of 'software ICs' is actually a reasonably accurate description of our technology.

Our research was conceived and developed independently of results in software engineering. As we have recently learned, our approach is an example of *domain analysis* (DA), a classical method in software engineering to identify reusable software modules ([Tra87, Pri87, Nei84]). DA relies on indepth studies of existing systems, published algorithms and structures to descern generic architectures for large classes of systems. A generic architecture is a template in which building-block modules can be plugged. Standardized interfaces are required to make blocks interchangable.

The primary difficulty with DA is the investment required to understand a domain in order to recognize building blocks. (Our research, for example, has been evolving for years). Not surprisingly, successful examples of DA are rather isolated.

In this paper, we explain our particular approach to domain analysis and how we have transformed our generic architecture of file management systems (FMSs) into a working prototype, an FMS synthesizer. The synthesizer enables a customized FMS to be assembled from prewritten components in minutes at virtually no cost. Producing a comparable FMS from scratch would require man-years of effort and hundreds of thousands of dollars. We have demonstrated the synthesizer at the 1988 ACM SIGMOD Conference and the 1989 International Conference on Software Engineering.

We begin with an overview of our approach, followed by a description and implementation of the FMS synthesizer. An earlier GENESIS prototype is described in [Bat88b].

2. The Approach

The most successful example of an existing building-blocks technology is the software libraries of scientific computing [Big87]. The main reasons for their success are: 1) the domain of numerical analysis is well-understood and its technology is reasonably static. These are the prime ingredients for standardization. 2) Standardization was possible because the domain was well-defined and small enough for experts to identify and agree upon the fundamental primitives.

We believe that mature software technologies are ripe for standardization. Using the semantics of a domain as a guide, it is possible for experts to define generic architectures for large classes of software systems. An architecture represents a standardized way to decompose different systems into primitive and reusable components. Our approach for standardization, both for architectures and component interfaces, is to study a large collection of systems, algorithms, and structures, to discern the simplest architecture and simplest interface that covers that class. The result is a blueprint for a building-blocks technology for software system construction.

This approach is outlined below. Validating examples are presented in Section 3 and [Bat88b].

* This work was supported by the National Science Foundation under grant DCR-86-00738.

2.1 Step 1: Defining a Generic Architecture

An architecture expresses a standardized way to modularly decompose systems into reusable components; it is a template that is common to all systems belonging to a particular class. Different systems share the same architecture, but are described by different components or different compositions of components.

We define an architecture as a collection of independently-definable objects and their interrelationships. An *independently-definable object* (IDO) is an object whose implementation has no impact on the implementation of other objects. A building-block is an implementation of an IDO.

Very few decompositions of systems can serve as an architecture for a building-blocks technology. As an example, database text books explain the functionality of a DBMS by presenting a generic architecture. A common decomposition offers a storage manager and a query optimizer among the primary components of a DBMS (Fig. 2.1a). The storage manager is responsible for storing and retrieving data, and the query optimizer is responsible for rearranging compositions of storage manager operations in order to minimize their combined expected costs (i.e., the costs of processing a query).

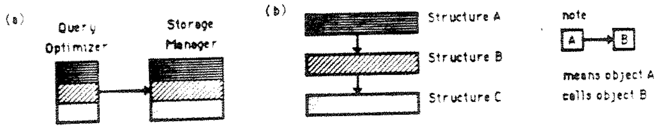


Figure 2.1 Independently Definable Objects

This is an obvious modularization as a storage manager and query optimizer are easily identifiable in virtually every DBMS. Unfortunately, they are implementation *dependent* objects. It is well-known that adding or removing storage structures in a storage manager must be accompanied by changes to the query optimizer. Similarly, one cannot replace the query optimizer of one DBMS with that of another because of an optimizer's dependency on the algorithms of its storage manager.

For an object to be implementation-independent, it must encapsulate an implementation detail and *all* algorithms that reference or maintain the consistency of that detail. The above example suggests that a storage structure, not a storage manager or query optimizer, would be an IDO. A storage structure object encapsulates all record retrieval, insertion, deletion, etc. algorithms associated with the structure *as well as* query optimization algorithms specific to that structure. A decomposition of a DBMS based on this abstraction is not intuitive, and is quite different than traditional views (Fig. 2.1b). The modules of Figures 2.1a-b cover exactly the same code; only the module interface boundaries are drawn differently. Storage structures are in fact IDOs [Bat88a].

Recognizing IDOs is difficult because it requires domain-specific expertise. A reliable method of IDO identification is to study a large class of existing systems, published algorithms, and structures to discern a generic system organization. By seeing enough examples, a steady-state architecture begins to emerge. (It is possible that there may be several distinct architectures, each describing a different class of systems). Assimilating quantities of knowledge for this purpose can be a formidable task.

Every object of an architecture is associated with a distinct class of modules. All modules are plug-compatible (for interchangeability), and each module is a different implementation of the object. Defining module interfaces is the next critical step in an architecture design.

2.2 Step 2: Defining Standardized Interfaces to Objects

Declaring an ad hoc interface to be a standard is the worst of all possibilities. A better way is to 1) identify the class of implementations that are to be supported, and 2) design the simplest interface that supports all implementations of the class. The greater the number of implementations, the more likely it is that the interface *captures fundamental properties of the class*. Such an interface is no longer ad hoc because it is justified by its demonstratable generality. We call this the *simplest common interface* (SCI) method of standardized interface design.

The internal data structures and data types that are required by the target class are standardized in the same way. All modules use a common set of variables and tables. Some modules require variables and columns *in addition* to the minimal set. As these variables are local to an implementation and are not visible to others, the required encapsulation of IDO modules is straightforward.

As an example, shadowing, page logging, and db-cache are three well-known page-based recovery algorithms. If one were to give each algorithm to a different implementor, three disparate interfaces would be designed even if all three used the same internal data structures. Algorithm interchangability would not be present. However, by defining a common interface for all three, interchangeability is guarenteed.

2.3 Principles and Techniques

Determining the underlying architectures and common interfaces of a collection of systems, algorithms, and structures involves traditional ER database design and modeling techniques [Teo86]. Hiding the implementation details of IDOs requires the encapsulation principles of object-oriented programming [Gol83]. Composing modules and defining modules with module parameters is the concept of polymorphic types [Lis77, Gog84, Vol85].

3. Building Blocks of File Management Systems

An integral component of a DBMS is a file management system (FMS). An FMS supports the storage and retrieval of data files and is responsible for database recovery in the event of crashes. An FMS is more primitive than a DBMS in that there is no query language (only a programming interface is provided), no query optimization, and no higher-level access structures (e.g., indices, transposition, etc.). An FMS provides the most primitive data storage and retrieval facilities of a DBMS.

Sections 3.1 and 3.2 explain FMS architecture schemes that we have developed and their building blocks. Readers who are familiar with FMS constructions should have no trouble following our discussions. (The nature of our work gives these discussions a tutorial flavor. Remember that the concepts are old; their packaging is new). Readers who are not interested in domain-specific details can skip to Section 3.3 where domain-independent techniques for implementing building-blocks are presented.

3.1 An Interface for FMSs

The objects that populate the interfaces of FMSs are records, files, volumes, primary keys, access keys, transactions, and buffer pools. Given the relationships below, basic operations on these objects are straightforward.

An FMS database is a set of files and volumes. A *volume* is a contiguous region on disk. A *file* is a collection of fixed-length or variable-length records. Multiple files can be stored in a single volume.

A file can be *unkeyed*, *single-keyed*, or *multi-keyed*. A multikeyed file has a single primary key which has been subdivided into distinct and nonoverlapping subkeys. Every record of a file has an *access key*, which is the FMS-defined storage location for the record. An access key could be a physical address or a primary key, depending on the underlying file structure. Records can be retrieved via file searching (using selection predicates) or by following pointers (access keys).

Users control buffer replacement and buffer allocation algorithms through the use of buffer pools. At first glance, it may seem surprising to find buffer pools at the FMS interface, especially when buffers themselves are not visible. However, the management of buffers within pools has a strong impact on FMS execution efficiency. Just as PROLOG needs cut symbols as hints to improve performance, buffer pools play a corresponding role in FMSs.

3.2 Single-User FMS Architectures and Their Building-Blocks

Different FMS architecture schemes arise as a result of a few major design decisions: whether or not to support database recovery from transaction, system, and media failures; whether or not to support concurrency, and choosing the size of the smallest lock granule (e.g., record, page, or file). Two architecture schemes for single-user FMSs are shown in Figure 3.1; they differ in the support or absense of database recovery facilities for transaction failures. We have implemented both. We describe each module class below, and list in Figure 3.2 the members that are presently available in our building-blocks library.

BLOCK is the class of modules that pack and unpack records in physical blocks. Records can be fixed-length or variable-length, and can be anchored (i.e., have a fixed physical address) or unanchored. A *node* or logical block is a sequence of records. NODE is the class of modules that map logical blocks to physical blocks. Among the classical node structures are: primary block only, primary block with unshared overflow, primary block with shared overflow, overflow only [Mar81], and a contiguous primary block structure [Lom89]. The records of a node can be unordered or maintained in primary key order.

The FILE class is a collection of file structure modules that map files to nodes. Among the classical file structures are B+ trees, isam, grid, heap, hash, and R-trees. The primary keys of files are implemented by modules of the KEY class. Among the primitive key types are integers and fixed-length strings.

VOLUME is the class of modules that provide for the creation, opening, closing, and deletion of individual volumes, and are responsible for physical space allocation within volumes. Blocks and space for individual records can be allocated sequentially or near a given location (to support block or record clustering).

BUFFER is the class of modules that manage pools of buffers as FMS resources and initiate physical block i/o. FMS users can create pools dynamically, and can select buffer allocation and replacement algorithms at run-time. Among the common allocation algorithms are global, volume-oriented, file-oriented, and process-oriented (e.g., Hot Set [Sac86]). Among common replacement algorithms are LRU, LRU-UNFIX [Eff84], least reference density [Sof80], and CLOCK [Eff84]. Physical block i/o (to raw disks or through operation system calls) is accomplished by the I/O module class.

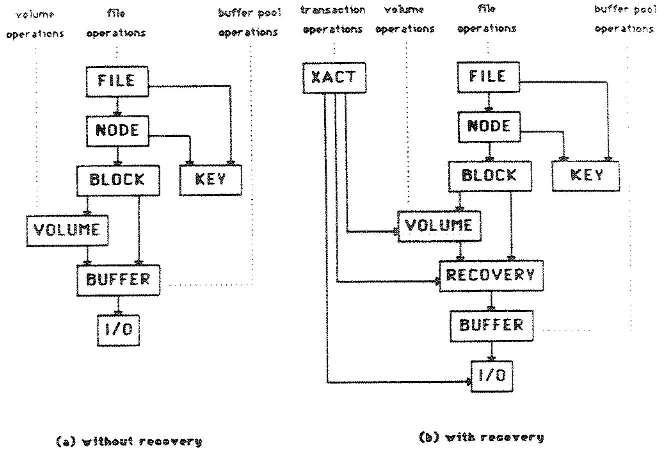


Figure 3.1 FMS Architectures

FILE	NODE	BLOCK	KEY	RECOVERY
B+ trees	Primary Only *	Fixed Unanchored	C string	null
Hash	Primary Unshared *	Fixed Anchored	Genesis String	Page Shadowing
Isam	Primary Shared *	Variable Unanchored	Integer	Page Logging
Heap	Overflow Only *	Variable Anchored	Long Integer	
Unordered				
Sequential	* unordered & ordered			
Grid				

XACT	BUFFER	VOLUME	I/O
Xact	Buffer	Volume	Unix Macintosh

Figure 3.2 Members of Module Classes Currently in our Library

RECOVERY is the class of modules that affect the recovery of individual volumes after a transaction failure. Standard recovery algorithms are shadowing, db-cache, and page-logging [Ber87]. Since a transaction may modify several volumes, the atomicity of a transaction must be realized by a two-phase commit. The modules of the XACT class accomplish this task. (Ignoring simple variations of two-phase commits, the XACT class effectively contains a single module).

Compositions of the above building-blocks yields file structures and file management systems. A file structure is a composition of a FILE module, the NODE modules which it references, and the BLOCK modules that the NODE modules reference. A file management system is a set of one or more file structures, a set of primitive KEY modules (referencable by all file structures), a VOLUME and BUFFER module, and a RECOVERY and XACT module (if recovery from transaction failures is supported).

3.3 Implementation

There are four different aspects to our implementation that make it novel: the software organization, debugging tools, software customization, and the FMS editor.

Software Organization. Given the fact that a class of modules share the same interface and the same internal data structures, it is not surprising that there is a considerable number of internal routines that are shared among modules.

As a first step, we carefully designed the modules of each class to maximize the number of common routines, thereby minimizing the total volume of code that needed to be written.

Another factor in software organization is whether or not an FMS will contain exactly one module of a class. For example, a FMS always has precisely one BUFFER module, one VOLUME module, and a single XACT and RECOVERY module (if transaction recovery is supported). Multiple FILE, NODE, BLOCK, and KEY modules will typically appear in every FMS. Different organizations are dictated for each case.

Our organizational concept is a **manager**, which is the packaging of common routines across one or more modules. A manager presents the interface of the module class. Thus, there is a BLOCK manager, a NODE manager, etc. Each operation of a manager is realized by a switch statement, where cases are algorithms that implement module-specific actions. For example, the ADVANCE operation in the FILE manager has the following organization:

```
ADVANCE( F )
{     /* common entry code */
      switch ( file_type_of( F ) ) {
      case BPLUS:        BPLUS_ADVANCE( F );
      case ISAM:         ISAM_ADVANCE( F );
      ...
      case GRID:         GRID_ADVANCE( F );
      };
      /* common exit code */
};
```

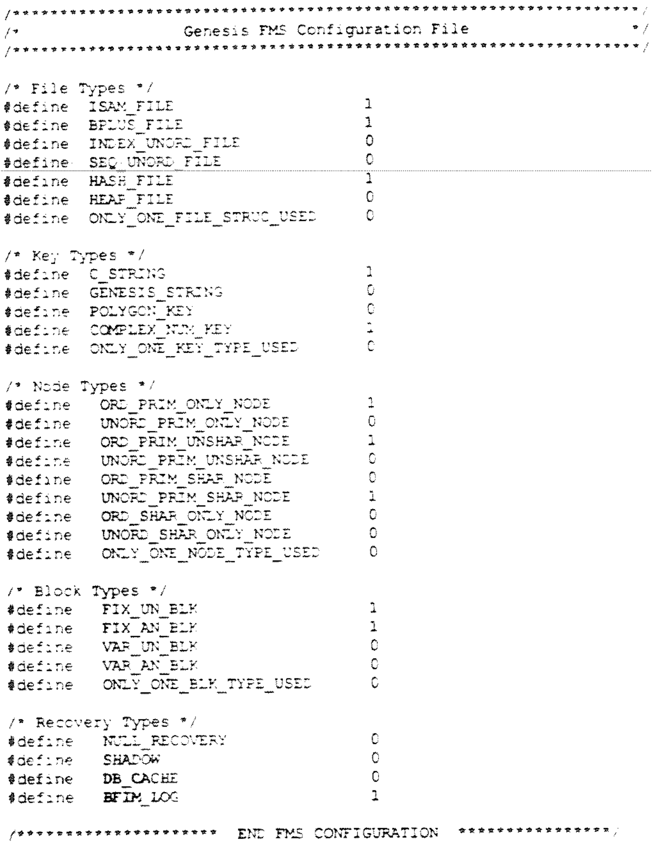
That is, entry code that is shared by all ADVANCE routines preceeds module-specific routines, which are followed by common exit routines. When ADVANCE is executed, the structure of the file determines the module-specific routine to call (e.g., if F is a BPLUS tree, then the BPLUS_ADVANCE algorithm is called). This same organization applies to all other operations of a manager's interface.

The organization of a manager is slightly different in the case that only a single module of a class will ever appear in a FMS. No switch statements are used. If there are n modules, n+1 source files are created. One contains source code that are common to all modules, and the remaining have source code specific to a module.

Software Development. A nontrivial fraction of the time it takes to develop modules is spent writing drivers to test module correctness. An important advantage to having a common interface for a class of modules is the ability to define a single driver for the entire class. Not only does this reduce development time, but also provides a clean way to develop a standard battery of tests to evaluate new modules that may be added later.

Software Customization. FMS customization is achieved solely through the use of the C precompiler. A single file of #defines, called a **configuration** file, specifies the target FMS. A boolean #define is declared for each module to indicate its inclusion/exclusion in the target system. In the case of managers with switch statements, cases of switches are eliminated or retained depending on their inclusion in the target system. Routines that are unreferenced by selected modules are also trimmed. As a last bit of optimization, we eliminate the switch statement (i.e., the 'switch ( ) { case : }' source code) if only a single module is to be retained. This too is accomplished through the declaration of a boolean #define.

In the case of managers that contain precisely one module, the common source file contains #includes for each of the module-specific files. At compile time, the common source file and the designated module-specific source file are merged. Source files of unreferenced modules are excluded. Figure 3.3 shows a configuration file.



```
/*************************************************************/
/*                 Genesis FMS Configuration File            */
/*************************************************************/

/* File Types */
#define  ISAM_FILE                    1
#define  BPLUS_FILE                   1
#define  INDEX_UNORD_FILE             0
#define  SEQ_UNORD_FILE               0
#define  HASH_FILE                    1
#define  HEAP_FILE                    0
#define  ONLY_ONE_FILE_STRUC_USED     0

/* Key Types */
#define  C_STRING                     1
#define  GENESIS_STRING               0
#define  POLYGON_KEY                  0
#define  COMPLEX_NUM_KEY              1
#define  ONLY_ONE_KEY_TYPE_USED       0

/* Node Types */
#define   ORD_PRIM_ONLY_NODE          1
#define   UNORD_PRIM_ONLY_NODE        0
#define   ORD_PRIM_UNSHAR_NODE        1
#define   UNORD_PRIM_UNSHAR_NODE      0
#define   ORD_PRIM_SHAR_NODE          0
#define   UNORD_PRIM_SHAR_NODE        1
#define   ORD_SHAR_ONLY_NODE          0
#define   UNORD_SHAR_ONLY_NODE        0
#define   ONLY_ONE_NODE_TYPE_USED     0

/* Block Types */
#define   FIX_UN_BLK                  1
#define   FIX_AN_BLK                  1
#define   VAR_UN_BLK                  0
#define   VAR_AN_BLK                  0
#define   ONLY_ONE_BLK_TYPE_USED      0

/* Recovery Types */
#define   NULL_RECOVERY               0
#define   SHADOW                      0
#define   DB_CACHE                    0
#define   BFIM_LOG                    1

/********************  END FMS CONFIGURATION  ***************/
```

Figure 3.3 An Example Configuration File

Customization Package. We developed a FMS editor running on Macintoshes to graphically demonstrate module (i.e., parameterized type) compositions and to generate configuration files. Figures 3.4 and 3.5 show the windows used for specifying file structures and file management systems.

A file structure is specified by linking together boxes that represent FILE modules, NODE modules, and BLOCK modules. An oval represents a choice to be made, and its label specifies the module class. Users are presented with the window shown in Figure 3.4a. Clicking the mouse causes a pop-up menu to appear listing all FILE modules that can be selected (Fig 3.4b). By choosing a particular module, the oval converts into a box labeled with the name of the selected module, plus additional ovals hanging from it indicating the NODE parameters of that FILE module (Fig. 3.4c). The same procedure is repeated to fill in the NODE parameters, and in turn, their BLOCK parameters. Figure 3.4d shows a fully specified file structure which has been given the name 'my.isam'.

An FMS is specified by selecting a set of file structures (created above), a set of primitive KEY types, a RECOVERY module, and a BUFFER module. Users are presented with the window shown in Figure 3.5a. The two scrollable subwindows contain the sets of file structures and KEY types already selected. By clicking the ADD button, a scrollable dialog box appears containing either all KEY types or all file structure types that are present in the GENESIS library (see Figure 3.5b). Entries are placed into subwindows by selecting them

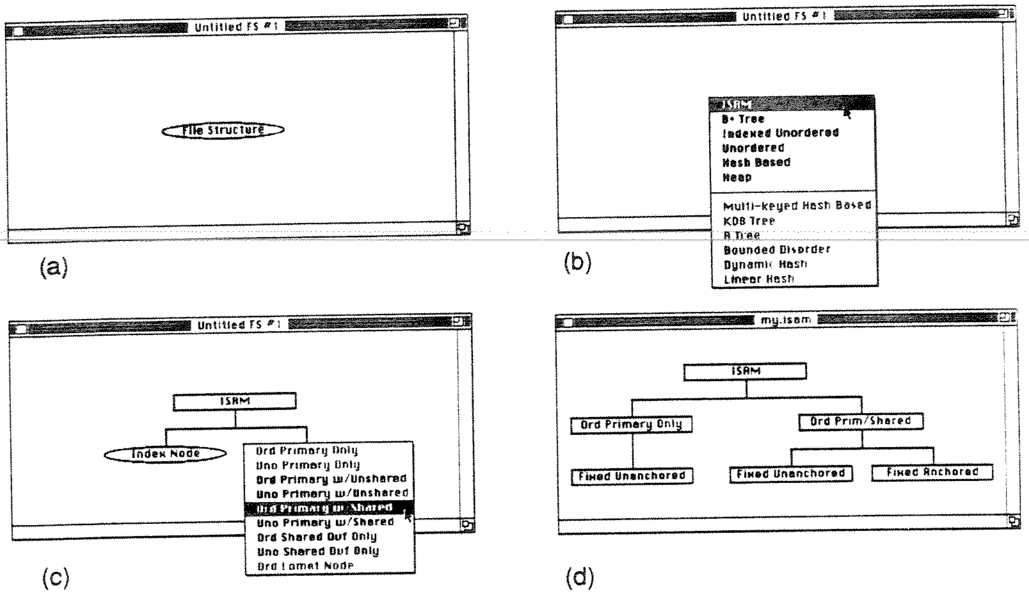


Figure 3.4 File Structure Design Sequence

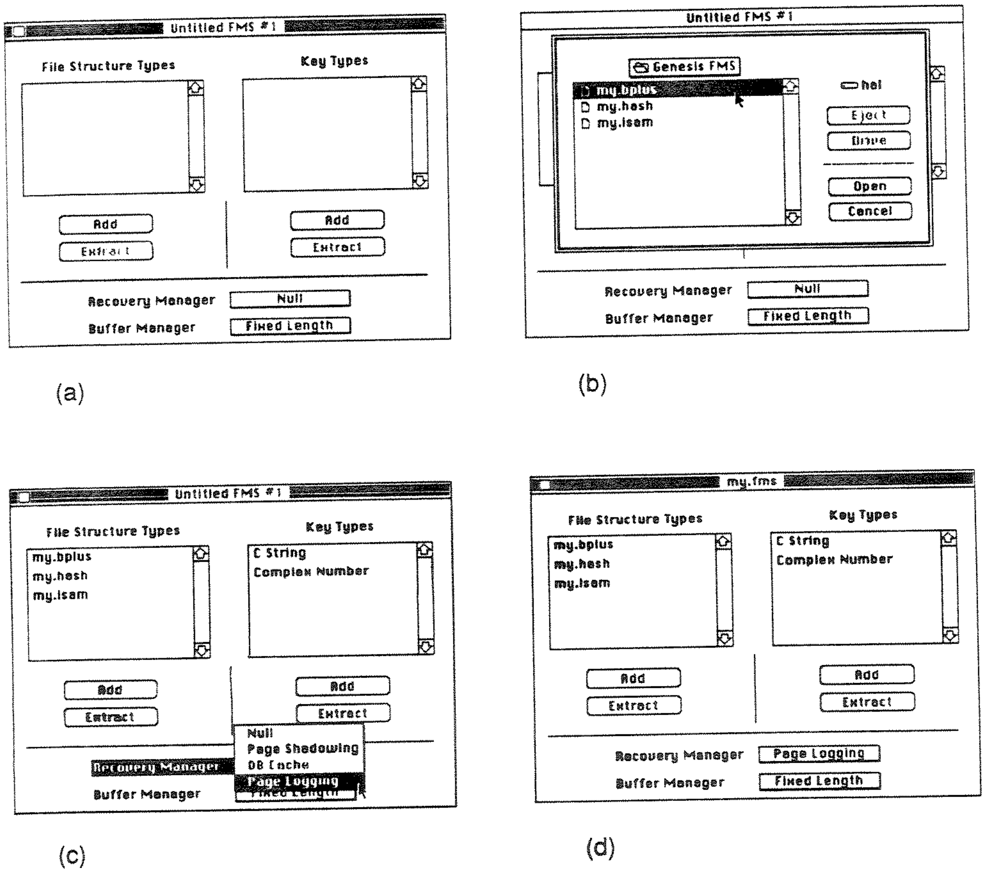


Figure 3.5 FMS Design Sequence

from the dialog boxes and are removed via the EXTRACT button. RECOVERY modules and BUFFER modules are chosen by clicking the RECOVERY or BUFFER boxes to obtain the menu of RECOVERY or BUFFER modules that are supported (Fig. 3.5c). Changes are made by overriding previous selections. Figure 3.5d shows a fully specified FMS named 'my.fms'.

Other features of the FMS Editor are: pull-down help menus describing each module, a statistics window which shows the size of each module class in lines of code, and the number of lines currently selected by a target FMS, and commands to generate configuration files, save, recall, and rename file structure definitions and FMS definitions.

It takes approximately fifteen minutes to define a FMS using the Editor and to produce its object code via compilation. In contrast, building an identical FMS from scratch would take several man years of work.

3.4 Performance and Future Work

Future work remains in three areas. 1) We are now extending our FMS architectures to handle concurrency control, recovery from system and media failures, and the storage and retrieval of of long records [Bar89]. 2) The primary goal of GENESIS is to demonstrate a building-blocks technology for both FMSs and DBMSs. We are now in the process of building (again) a DBMS synthesizer [Bat88b] and expect the prototype to be completed in Fall 1989. 3) A thorough performance evaluation and tuning of our software is a major task that remains. Preliminary performance measurements on untuned B+ trees show that records can be bulk loaded at a rate of 1.6 million per hour, retrieved at 2.5 million records per hour, and randomly inserted at 225 thousand records per hour on a dedicated Sun 3/160.

4. Lessons and Experience

Our research confirmed two facts that we anticipated from the beginning: 1) a FMS synthesizer is a great prototyping tool for generating alternative FMSs, and 2) module reusability enormously reduces technology reinvention. We did not forsee problems in design standardizations and, more importantly, the misconceptions people have about our work.

Design Issues

- Standardized interfaces and architectures work if they are general enough. Unfortunately, 'general enough' can not always be precisely defined. Undergeneralizing occurs when one considers a class of algorithms or structures that are too restrictive. The introduction of new algorithms and new structures may cause 'standard interfaces' to evolve, thereby triggering potentially costly module revisions. Overgeneralizing also has drawbacks. By mistakenly grouping algorithms and structures that really perform very different functions, designing SCI interfaces for such classes leads to untenable situations (e.g., operation semantics become module-dependent). Finding a balance required more design iterations than we had expected.
- The time and funds needed to develop a FMS synthesizer is several times that required to build a single FMS from scratch. The design phase is more involved and a longer implementation phase is needed to produce a library of critical mass [Big87]. We had anticipated a much shorter development time.

Misconceptions

- Developing a building-blocks technology is not a matter of listing algorithms and structures and just choosing what you want. The intellectual challenge is understanding the domain well enough to separate implementation issues into orthogonal units (IDOs). *For systems as complicated as DBMSs, it can easily take an expert several years to design and verify building-block architectures. We foresee no quick and easy solutions.*
- Our technology cannot easily assemble *any* system, but only systems in family for which it was designed. Enlarging the family has various degrees of difficulty. Adding a new module to a class is not difficult. Adding a new operation to a class involves upgrades to many modules. (This can be simple or hard; it depends on the operation. Usually, such extensions are trivial). Adding a new class is a significant change and usually requires a major rewrite. In such a case, we doubt if available building-blocks will help. As a rule, however, adding fundamentally new classes is unusual.
- Our building-blocks do not simplify difficult technical problems or provide new solutions to problems. Adding random sampling capability to an FMS is a classical example. How this is to be done in *any* FMS is the subject of much research and speculation. What algorithms should be used isn't yet clear [Olk88]. Our technology only offers a framework in which to explore solutions to a problem in a general setting. *The framework itself is not a solution; it is only a platform on which to implement a class of previously identified solutions.*

5. Conclusions

We have explained an approach to domain analysis that we have followed in the development of a building-blocks technology for FMSs and DBMSs. We validated the approach through prototyping, and we are working toward a more precise definition of the principles and design processes which have implicitly guided our work. We believe that mature software technologies are ripe for standardization, and that building-block technologies are possible using the techniques that we have referenced and outlined.

Recognizing building-blocks is very difficult for systems as complicated as DBMSs. The intellectual challenge is identifying IDOs, and showing how known algorithms and structures can be explained as combination of IDO implementations. The FMS synthesizer we have presented has taken at least three man-years of work to design and build, but it also presents a FMS prototyping technology that is unequalled in DBMSs today. We believe our work provides yet another step toward making software reusability a reality.

Acknowledgements. We thank John Werth for his helpful comments on an earlier draft.

6. References

[Bat88a] D.S. Batory, 'Concepts for a Database System Synthesizer', *ACM PODS* 1988, 184-192.

[Bat88b] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, 'GENESIS: An Extensible Database Management System', *IEEE Trans. Software Engr.*, November 1988.

[Bar89] J.R. Barnett and D.S. Batory, 'A Uniform Mechanism to Support Long Fields and Nested Relations in DBMSs', to appear in *Proc. Hawaii Int. Conf. System Sciences*, 1989.

[Ber87] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Big87] T. Biggerstaff and C. Richter, 'Reusability Framework, Assessment, and Directions', *IEEE Software*, 4,2 (March 1987), 41-49.

[Car86] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita, 'The Architecture of the EXODUS Extensible DBMS', *Workshop on Object-Oriented Database Syst.*, 1986, 52-65.

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

[Gog84] J. Goguen, 'Parameterized Programming', *IEEE Trans. Software Engr.*, SE-10,5 (September 1984), 528-543.

[Gol83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[Lee88] M.K. Lee, J.C. Freytag, and G.M. Lohman, 'Implementing an Interpreter for Functional Rules in a Query Optimizer', *VLDB* 1988, 218-229.

[Lis77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, 'Abstraction Mechanisms in CLU', *Comm. ACM*, 20,8 (Aug. 1977), 564-576.

[Lom89] D. Lomet, 'A Simple Bounded Disorder File Organization with Good Performance', to appear *ACM Trans. Database Syst.*

[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.

[Nei84] J.M. Neighbors, 'The Draco Approach to Constructing Software from Reusable Components', *IEEE Trans. Software Engineering*, Sept. 1984, 564-574.

[Olk88] F. Olken, private correspondence.

[Pri87] R. Prieto-Diaz, 'Domain Analysis for Reusability', *Proc. COMPSAC 1987*, 23-29.

[Sac86] G.M. Sacco and M. Schkolnick, 'Buffer Management in Relational Database Systems', *ACM Trans. Database Syst.* 11,4 (Dec. 1986), 473-498.

[Sof80] Software AG of North America, Inc., *ADABAS: Effective Data Base Management for the Corporate Environment*, Reston, Va., 1980.

[Sto86] M. Stonebraker and L. Rowe, 'The Design of POSTGRES', *ACM SIGMOD* 1986, 340-355.

[Teo86] T.J. Teorey, D. Yang, and J.P. Fry, 'A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model', *ACM Comp. Surv.*, 18,2 (June 1986), 197-222.

[Tra87] W. Tracz, 'RMISE Workshop on Software Reuse Meeting Summary', in *Software Reuse: Emerging Technology*, W. Tracz, ed., IEEE Computer Society Press, 1988.

[Vol85] D.M. Volpano and R.B. Kieburtz, 'Software Templates', *Int. Conf. Software Engineering* 1985, 55-60.

DaTE: The Genesis DBMS Software Layout Editor *

D.S. Batory and J.R. Barnett
Department of Computer Sciences
The University of Texas
Austin, Texas 78711

1. Introduction

DBMSs are complex software systems that are notoriously difficult to build. Extensible database systems were conceived to ease the burden of DBMS construction. A number of different approaches to extensibility have been proposed and prototyped [Car88, Haa89, Sto86]. Among them, the Genesis approach is distinguished as a software building-blocks technology [Bat85-91]. Its premise is that complex software systems can be constructed from prefabricated components in minutes at virtually no cost.

Genesis 2.0 became operational in November 1989. Our objectives were achieved: customized relational DBMSs in excess of 50K lines of C could be specified and their executables produced within a half hour. In this paper, we examine an integral tool of this construction process: the Genesis software layout editor (DaTE). [1]

DaTE is a graphical design tool used by database system implementors to specify the construction of customized DBMSs and FMSs (file management systems) as compositions of available software components. DaTE embodies a stratified top-down design methodology, where implementation details, ranging from the selection of nonprocedural data languages to the packaging of records into physical blocks, are captured through component assemblies. File structures, storage systems, network DBMSs, and relational DBMSs are contiguous regions of the design space encompassed by DaTE. Transcripts of a DBMS or FMS design can be output which, when compiled with the Genesis library, yields the executables of the target system. [2]

In this paper, we examine the features of DaTE. We explain its graphics, how it captures the domain model of Genesis, and how sophisticated DBMSs can be defined in minutes as compositions of components. Similar to problems encountered in hardware layout editors, indescriminate combinations of components may yield systems that cannot possibly work.

* This work was supported by the National Science Foundation under grant DCR-86-00738.

[1] DaTE is an abbreviation for *D*atabase *T*ype *E*ditor.

[2] As of mid-1991, DaTE is 22K lines of C that runs on a MacIntosh II. We are in the process of rewriting DaTE to run in X-window environments.

Design rule checking, an integral (but effectively unseen) feature of DaTE, ensures that all systems that can be specified are technically correct. We will study incorrect DBMS designs, define design rules that prohibit such designs from arising, and present efficient algorithms that are used in DaTE to enforce design rules.

To understand DaTE and the algorithms that we develop, it is important for readers to be familiar with the conceptual abstractions on which Genesis is based. We begin our discussions with a brief overview of these ideas.

2. An Overview of Genesis

Software systems, and DBMSs in particular, are becoming progressively more complex and more costly to build. While it was possible in the past to build new systems from scratch, this is a luxury that the software industry (and again, the database industry in particular) will soon no longer be able to afford.

Genesis is a project whose goal was to demonstrate that mature and well-understood software technologies could be standardized as libraries of prefabricated and reusable components, and that customized software systems of considerable complexity could be assembled in minutes by gluing available components together. A general model of large scale software was conceived, and the domain of relational database technology was expressed in its terms [Bat85-91]. The distinction between the Genesis approach to extensibility and the Exodus and Starburst approaches, for example, is the building-block/component approach to system construction.

The Genesis model of software development is straightforward. A *domain* (e.g., relational technology) is expressed as a set of *components* organized into *realms*. A realm R is associated with a particular interface. All components of a realm share the same interface, which means all components are plug-compatible and interchangeable. Realms R and S shown below each have three components each.

$$R = \{ a, b, c \}$$

$$S = \{ d[x{:}R], e[x{:}R], f[x{:}R] \}$$

A component may have parameters, meaning that it needs services from lower-level components. How lower-level services are implemented is specified via component parameter instantiation. As an example, each of the components of realm S has a single parameter of type R. (The notation x:R means x must belong to realm R). In principle, any component in R could provide services for any component in S.

A software system is a type expression. Three systems constructed from the above realms are shown below:

$$\begin{aligned} \text{system1} &= d[a] \\ \text{system2} &= d[c] \\ \text{system3} &= f[c] \end{aligned}$$

Each of these systems provide the same functionality, but because they are composed from different components, they may exhibit different performance characteristics.

Quantification of software *reuse* has traditionally been difficult [Big89]. In the Genesis model, recognizing reuse is simple. Whenever two or more systems (type expressions) reference the same component, that component is being reused. In the above systems, components d and c are being reused.

A fundamental feature of the model is the concept of reflexive components. Reflexive components are unusual in that they can be composed in virtually arbitary ways. In the

Genesis formalism, this means that components of some realm T have parameters of type T:

$$T = \{ m[x:T], n[x:T], ... \}$$

Thus, compositions m[n[ ]] and n[m[ ]] are possible. Unix file filters are classical examples of reflexive components.

Extensibility is achieved through the addition of new components in realms, and (less frequently) through the addition of new realms. In principle, if realm libraries are well-stocked, all components needed for a target system are present. It should then be possible to specify the target system and have it assembled (given its specification) in minutes. In the case that one or more needed components are not present in the realm libraries, they must be written. However, since one relies on the availability of prewritten components, the overhead for system development is considerably reduced. From our experience, an individual component takes somewhere between two weeks to three months to implement, depending on its complexity. In general, component reuse reduces the time for system construction by approximately an order of magnitude.

Two basic problems arise in developing software system in this manner. First, complex systems often correspond to complicated type expressions that are difficult to read and interpret. For this reason, a graphical representation of designs is needed. Second, not all combinations of components are semantically meaningful. Design rules, which preclude certain compositions of components and guarentee the design of correct (i.e., operational - not necessarily efficient) systems, must be specified and enforced. DaTE is the first attempt to develop a graphical layout editor that is based on the above model and that enforces design rules. The rest of this paper explains the mechanics and operation of DaTE.

3. The Genesis Software Layout Editor (DaTE)

Version 2.0 of Genesis supports twelve distinct realms, each with different numbers of components. The table below briefly explains each.

Class	Description
model	data languages (e.g., sql and quel)
link	join algorithms and linkset implementations (e.g., nested loop , ring list)
file mapping	abstract-to-concrete file mappings (e.g., indexing, compression)
file storage	primitive file structure algorithms (e.g., B+ trees, heaps)
logical block	logical-to-physical block mappings (e.g., overflow only, shared overflow)
physical block	fixed-length and variable-length record blocking algorithms
special operation	operations used in query processing (e.g., cross product, sort)
data type	primitive data types that are referenced in schemas (e.g., float)
recovery	volume recovery algorithms (e.g., before image logging, shadowing)
transaction	multivolume commitment protocols (e.g., two-phase commit)
buffer	buffer management algorithms
input/output	Genesis-to-operating-system mappings (e.g., Unix)

The graphics of DaTE reinforces the building-blocks paradigm of Genesis. DaTE depicts components as boxes and parameters as ovals. Components M and N are shown in Figures 3.1a-b. M is unparameterized; N has parameters X and Y.

DaTE distinguishes two different types of component compositions: *systems* and *subsystems*. A system is a stand-alone application. DaTE supports the definition of two types of systems: DBMSs and FMSs. We explain their difference in Section 3.4. A subsystem, in contrast, is a coherent portion of a system that is not stand-alone. Four progressively more

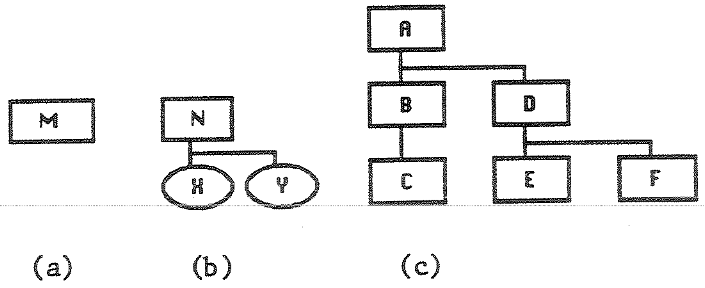


Figure 3.1 DaTE Graphics

complex types of subsystems are definable by DaTE: file structure, storage, network, and relational. Their differences are explained in Section 3.3.

A subsystem is depicted as a rooted graph whose vertices represent either components or complete systems. A subsystem is *complete* if it has no unbound parameters (i.e., ovals). The subsystem in Figure 3.1c is an example. Complete subsystems are treated as unparameterized components by DaTE.

A central concept in Genesis is conceptual-to-internal mappings of files. Every conceptual file can be mapped to one or more internal files. Precisely one internal file is *dominant*; the remaining are *subordinate*. The critical properties of dominance, as prescribed in [Bat85], is that records of a dominant file are always in 1-to-1 correspondence with conceptual records; this need not be the case for subordinate files. Moreover, the address of an abstract record is always the same as the address of its dominant concrete record.

As an example, consider the component that provides secondary indexing. It maps a relation to an inverted file, which consists of a data file plus zero or more index files. The data file is the dominant concrete file of this mapping, as each data file record is in 1-to-1 correspondence with tuples of the relation. Moreover, the identifier of a tuple is indistinguishable from the address of the tuple's dominant concrete record. Index files and index records are not dominant.

DaTE captures the notion of dominance in its vertical-column arrangement of nodes of a subsystem. The left-most column of every subsystem, starting at the top node and moving downward, defines the sequence of components through which a conceptual file is mapped to its dominant internal counterpart. The remaining vertical columns define sequences of components through which subordinate files are mapped. As an example, the subsystem in Figure 3.1c has three columns (#1: A,B,C; #2: D,E; #3: F). Component A maps an abstract file to a single dominant concrete file and zero or more subordinate concrete files. The dominant concrete file is mapped by the vertical column of components beginning with B; subordinate concrete files are mapped beginning with component D.

The most powerful concept in DaTE is the *software bus*. It is an abstract construct that allows multiple components to occupy the same position in a subsystem. Software busses arise in two rather different circumstances. An *implementation bus* is used to list alternative methods of implementing an object. A file bus that lists **grid** and **bplus**, for example, enables a file to be implemented by either a grid structure or a B+ tree. On the other hand, a *feature bus* lists disparate attributes that a target subsystem is to exhibit. For instance, a data type bus lists the data types that a target DBMS or FMS is to support. Whether a bus is implementation-oriented or feature-oriented is evident from the context.

A software bus is depicted as a scrollable window. The bus in Figure 3.2a lists the components (or subsystems) **bplus** and **isam**: Clicking the **Add** button admits new entries onto a bus. An entry (such as **isam**) is deleted by clicking it and choosing the **Remove** option from the displayed popup menu (Fig. 3.2b).

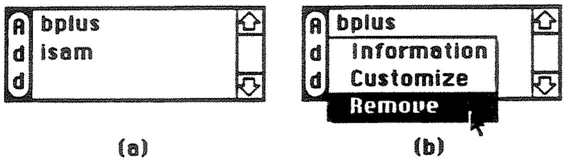
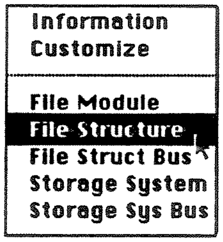


Figure 3.2 A Software Bus

Systems are classified according to their type (since they are type expressions). That is, there is the class of file structure subsystems, storage subsystems, relational subsystems, and so on. For simplicity, DaTE restricts entries of a software bus to belong to a single type/realm. Thus, all entries of a bus are storage subsystems, or all are link components, etc.

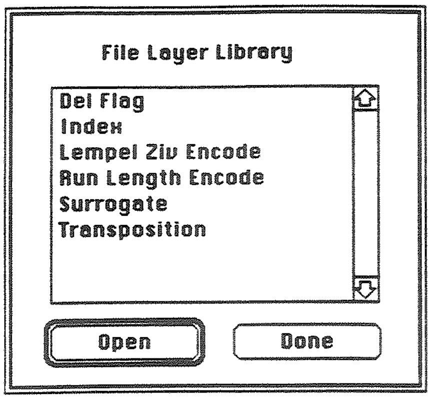
3.1 Parameter Instantiation and Editing Rules

Parameter instantiation in DaTE is accomplished by clicking an oval. The standard response is the display of a popup menu, like the one below:



Selecting **Information** displays a help window associated with the selected item. **Customize** lists the customizable options of a component, and allows options to be enabled or disabled. (For example, if the **Index** component supports several algorithms for processing queries using inverted lists, customization would allow an implementor to select only the algorithms that need to be included in a target system).

Selecting an entry below the dotted line causes a scrollable **library window** to be displayed. The members of the library are legal components or subsystems that can instantiate the selected parameter. How such components or subsystems are determined is the subject of Design Rule Checking, the topic of Section 4. A library window for File components is shown below:



Clicking the desired entry in the window triggers parameter instantiation.

Occasionally, parameters are bound incorrectly (or better choices are later discovered). Rebinding is accomplished by clicking the component to change and by choosing a substitute from the displayed library window. As DaTE imposes a top-down design methodology, all hierarchical bindings of the original component may be erased as they no longer apply to the new selection. DaTE tries to save such bindings whenever possible.

3.2 Subsystems

As mentioned earlier, a subsystem is a rooted graph of primitive components and complete subsystems. Each subsystem (recall the four types: file structure, storage, network, and relational) is defined within a special window that can be named and saved for later reference. In the following sections, we show how each of these subsystems can be created. We begin with the simplest of DaTE subsystems: file structures.

3.2.1 Creating a File Structure Subsystem

A file structure is a composition of components that provide the most elementary file storage and retrieval capabilities needed for DBMS operation. A file structure is a composition of three distinct types of components: FS (file storage), logical block (or node), and physical block.

A file structure is created in DaTE by pulling down the **File** menu from the menu bar, selecting **New**, and then **File Structure**. An empty window is then displayed (Fig. 3.3a):

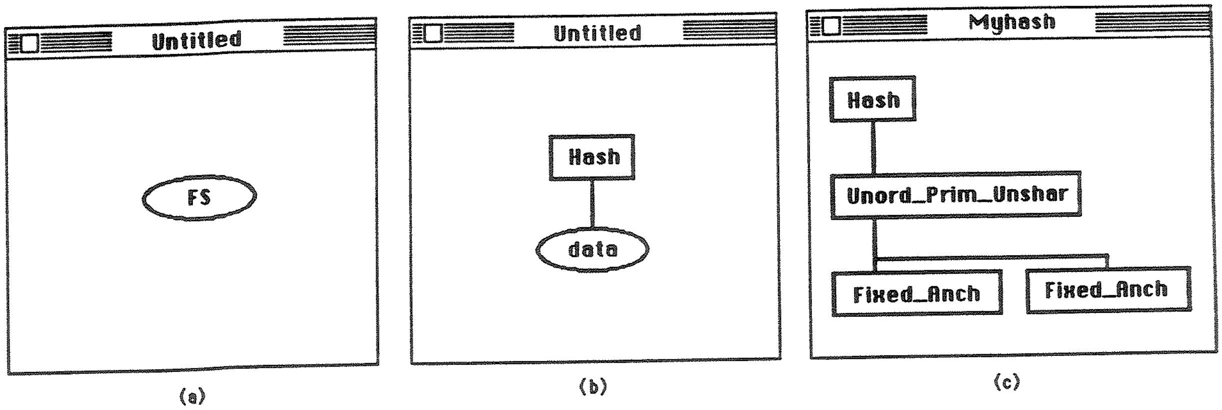


Figure 3.3 Building a File Structure Subsystem

The window contains a single **FS** oval, indicating that a file storage component must be specified. Clicking the oval causes a library window to appear that lists all FS components known to DaTE. Once an FS component is chosen, its box is displayed in the window, along with its logical block parameters (Fig. 3.3b). (Note that an FS component maps a file of records to logical blocks. The first or left-most parameter of the component specifies how data records are to be stored; a second, if present, specifies how index records are stored. In the above example, the **Hash** component does not generate index records).

Logical block parameters are instantiated in the same way, i.e., by clicking and selecting components from library windows. Each logical block component has one or more physical block parameters. (The left-most parameter specifies how records are packaged in a primary block; a second, if present, specifies the packaging of records in overflow blocks).

Figure 3.3c shows a three-level graph that defines the **Myhash** subsystem. A **Hash** file storage algorithm is used, whose logical blocks are implemented by the **Unord_Prim_Unshared** component (i.e., unordered records stored in a primary block with unshared overflow), and primary and overflow blocks are **Fixed_Anch** (i.e., fixed-length records with anchored physical addresses).

Table 1 lists the file storage, logical block, and physical block components presently available in the Genesis library. Over sixty distinct file structure subsystems can be created with them.

File Storage Components	Logical Block Components [3]	Physical Block Components [4]
bplus	unordered primary block only	fixed anchored
grid	unordered shared overflow	fixed unanchored
hash	unordered unshared overflow	variable anchored
heap	unordered overflow only	variable unanchored
indexed unordered	ordered primary block only	
isam	ordered shared overflow	
sequential unordered	ordered unshared overflow	
	ordered overflow only	

Table 1. Primitive Components of File Structure Subsystems

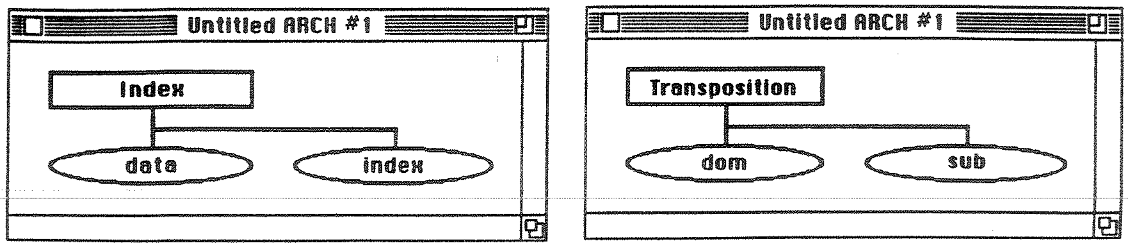
3.2.2 Creating a Storage Subsystem

A *file mapping component* maps an abstract file to precisely one dominant concrete file and zero or more subordinate concrete files. A *storage subsystem* is a composition of file mapping components that terminate with references to complete file structure subsystems.

A storage subsystem begins with the creation of an empty storage subsystem window. As classical examples of file mapping modules, the windows in Figures 3.4a-b show the result of selecting the indexing and transposition components. **Index** maps an abstract file to a dominant data file and zero or more subordinate index files. The data file implementation is specified by parameter **data** and the index file implementation by parameter **index**. Similarly, **Transposition** maps a file to a series of concrete subfiles, one dominant subfile and zero or more subordinant subfiles. Their implementations are given by parameters **dom** and **sub**. The file mapping components currently available in the Genesis library are listed in Table 2.

[3] A logical block is a sequence of records, which can be *unordered* or be maintained in primary key *order*. If logical blocks have a bounded record capacity, the records can be stored in a single physical block (*primary block only*). If capacity is unbounded, overflow records can be stored in physical blocks dedicated to the logical block (*unshared overflow*) or in physical blocks that are shared among different logical blocks (*shared overflow*). *Overflow only* is a logical block implementation that is simply a chain of overflow records. The combinations of ordering and packaging yields eight distinct implementations of logical blocks.

[4] Records are either *fixed* length or *variable* length. They can be assigned permanent physical addresses (*anchored*) or not (*unanchored*).



(a) secondery indices

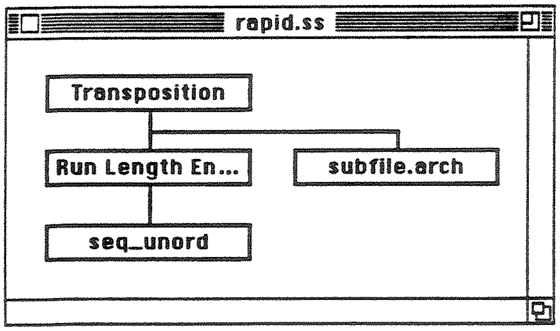
(b) transposition

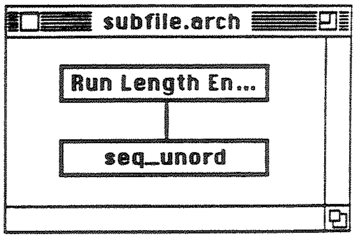
Figure 3.4 File Mapping Components

file mapping components	link components	data model components
delete flag	(block) nested loop	quel
(secondary) index	pointer array	sql
run-length encoding	ring list	
surrogate	sort merge	
transposition		
ziv-lempel encoding		

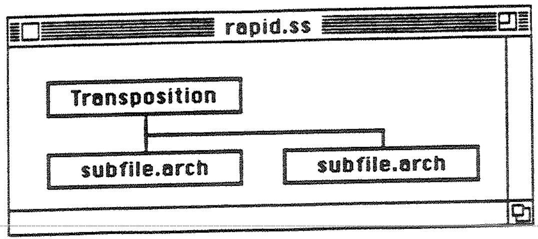
Table 2. Primitive Components of Non-File Structure Subsystems

Consider a storage subsystem that approximates Rapid, a statistical DBMS [Tur79]. Rapid mapped schema-defined files to transposed files, where each column was run-length compressed before being stored in a sequential-unordered file structure. This subsystem is defined in two windows: **rapid.ss** and **subfile.arch**. **rapid.ss** maps a schema-defined file to its dominant internal counterpart. **subfile.arch** maps subordinate files to their internal counterparts.



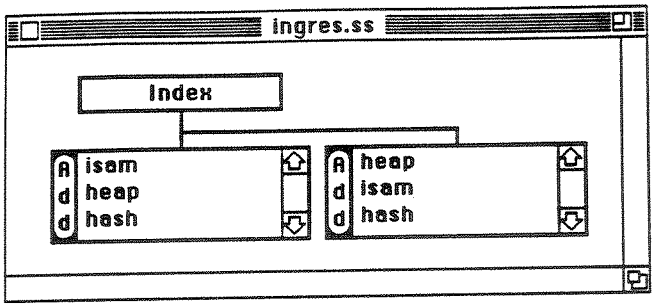


While it may seem odd not to have **Transposition** call **subfile.arch** twice (as implementations of both dominant and subordinate files are identical), DaTE permits only one subsystem reference per dominant (vertical) mapping. (It turns out that permitting multiple subsystem references significantly increases DaTE's complexity without providing greater expressibility. We chose simplicity).



A composition that cannot be defined directly in DaTE....

As another example, consider the storage subsystem of University Ingres [Sto76]: it maps schema-defined files/relations to inverted files, where data files and index files can be selectively implemented by hash, heap, or isam structures. The multiplicity of implementation choices is captured by a pair of file structure busses. [5]



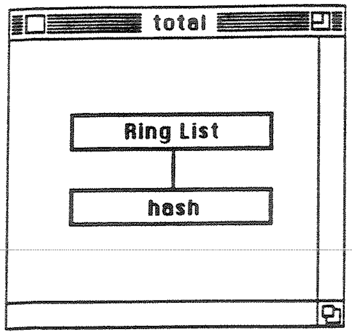
3.2.3 Creating Network and Relational Subsystems

A *network subsystem* is rooted by a link component or link bus. This component or bus specifies how links - i.e., relationships between files - are to be implemented. The sole parameter of a link component or link bus specifies how files are implemented, which may be expressed by a single file structure or storage subsystem, or a bus of file structure or storage subsystems.

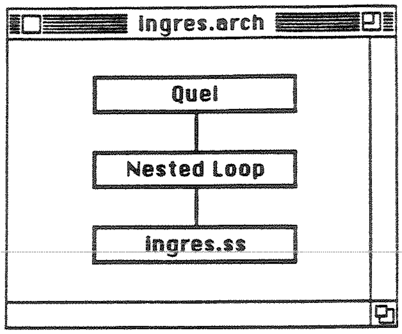
A *relational subsystem* is rooted by a data model component or data model bus. Data model components map a nonprocedural data language interface to a procedural network database interface. The sole parameter of a data model component or bus is the implementation of the links of the network database.

The figures below show a network subsystem used in the Total DBMS (i.e., no high-level data model; links are implemented by ring-lists and files are stored in hash structures), and the relational subsystem of University Ingres (i.e., Quel is the data language, nested loop implementations of links, and files are stored in the ingres storage subsystem).

[5] The last entry on a software bus is the default mapping. Thus, data files in **ingres.ss** default to hash-based structures if no storage structure directive is provided. (Such directives are part of Genesis database schemas). No other significance is attributed to the ordering of entries on a implementation bus.



(a) A model of Total



(b) A model of Ingres

The link and data model components currently available in the Genesis library are listed in Table 2.

3.3 Systems

A *system* is a composition of one or more subsystems with software busses listing supporting primitive components, such as data types, recovery, and special operations. FMSs and DBMSs are systems that can be defined and generated by DaTE. In the following sections, we illustrate how complicated FMSs and DBMSs are specified and generated.

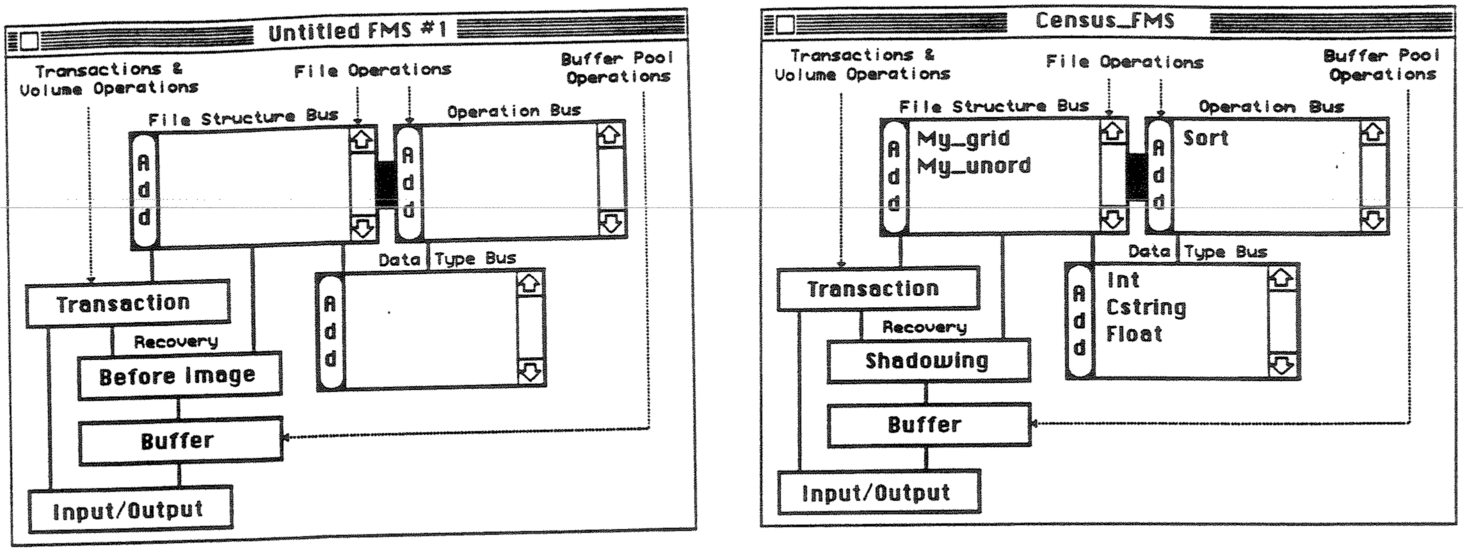
3.3.1 Creating a File Management System

A *file management system (FMS)* is the kernel of a DBMS. It provides elementary access methods, buffer management, and recovery capabilities necessary for DBMS operation. DaTE factors the design of an FMS into the selection of file structures, special operations, data types, and a recovery component.

An FMS is created in DaTE by pulling down the **File** menu, selecting **New**, and then **FMS**. An empty FMS window is then displayed (Fig. 3.5a). The above mentioned design decisions are entered onto three busses and a field. Each is labeled in small font as a prompt to the FMS implementor. Note that **Before Image** page logging is the default implementation of recovery. Also note that the **Transaction**, **Buffer**, and **Input/Output** classes are not customizable, as Genesis provides only a single component for each. This will change as other components become available.

Observe that the graphics of an FMS window documents the routing (via dotted lines) of user-issued operations. Volume and transaction operations are serviced by the **Transaction** components; buffer pool operations are handled by the **Buffer** components. File operations are processed by either file structures or special operations. Also note that an FMS is not a strict hierarchy of components (unlike subsystems), where all operations are transformed by components in a top-down manner.

A possible FMS for a census database is shown in Figure 3.5b. It provides **My_grid** and **My_unord** as primitive file structures. The **Sort** operation is included, along with the data types **Int**, **Cstring**, and **Float**. Page **Shadowing** is used for volume recovery. Table 3 lists the recovery, data type, and special operation components that are currently available.



(a) (b)

Figure 3.5 Empty and Completed FMS Windows

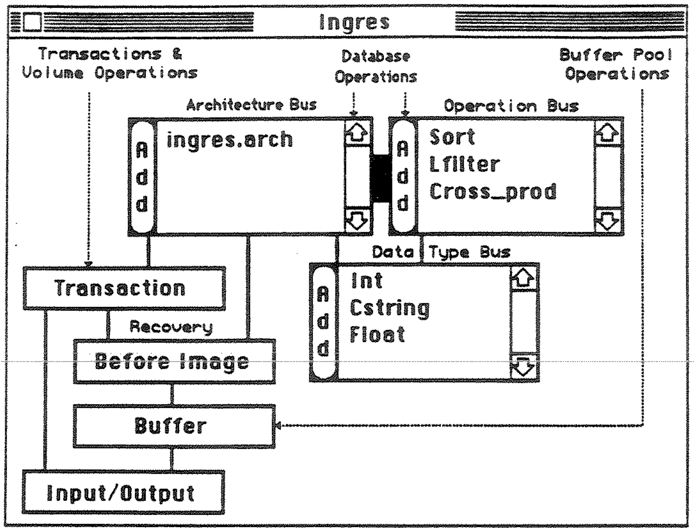
Recovery	Data Type	Special Operations
db cache	byte	cross product
before image logging	char	lfilter
null (no recovery)	cstring	cross product
page shadowing	double	
	int	
	float	
	short	
	vstring	

Table 3. Primitive Components of File Structure Subsystems

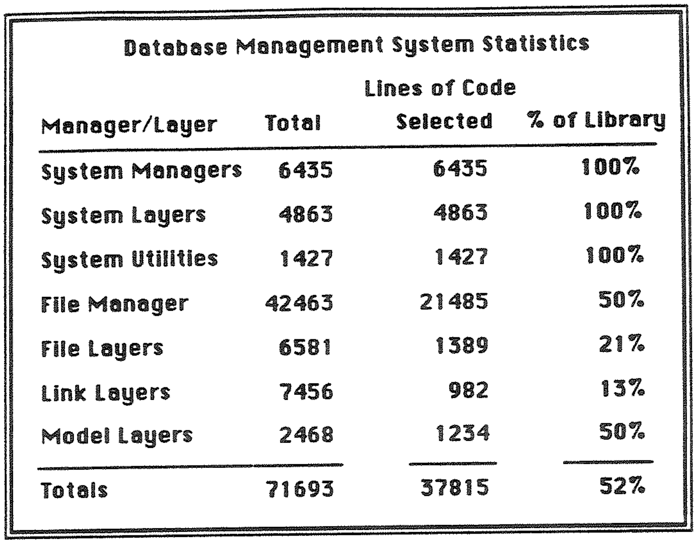
3.3.2 Creating a Database Management System

A DBMS is defined in the same manner as FMSs: subsystems, special operations, data types, and a recovery component must be specified. The only difference is that relational, network, and storage subsystems are referenced instead of file structures.

A DBMS window for our approximation of University Ingres is shown below: the subsystem is **Ingres.arch**; special operations are **Sort**, **Lfilter**, and **Cross_Prod**; data types are **Int**, **Cstring**, and **Float**; and recovery is handled by **Before Image** logging.



Statistics about the size of the generated DBMS can be obtained by pulling down the **Misc** menu and selecting **Statistics**. A screen similar to the one below will be displayed:



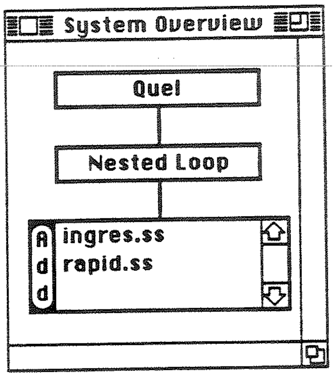
Database Management System Statistics

	Lines of Code		
Manager/Layer	Total	Selected	% of Library
System Managers	6435	6435	100%
System Layers	4863	4863	100%
System Utilities	1427	1427	100%
File Manager	42463	21485	50%
File Layers	6581	1389	21%
Link Layers	7456	982	13%
Model Layers	2468	1234	50%
Totals	71693	37815	52%

Of the 71K+ lines of code in the Genesis libraries, approximately 52 percent is referenced in our approximation to Ingres. (Lines that are unreferenced are not included when **Ingres** is compiled). A similar screen exists for FMSs. [6]

A DBMS typically supports only one subsystem. However, if one wants the *union* of several different subsystems (i.e., to have the capabilities of several individual DBMSs rolled up into a single DBMS), one can click multiple subsystems onto the **System Bus**. A composite subsystem is formed by collecting all distinct data models, links, and storage subsystems and placing them on their corresponding software busses. (Thus, it is possible for a Genesis-produced DBMS to support multiple data languages). A composite subsystem of a target DBMS can be viewed by pulling down the **Misc** menu and selecting **DBMS Overview** when the DBMS window is active. The union of **Ingres.arch** and **rapid.ss** is shown below:

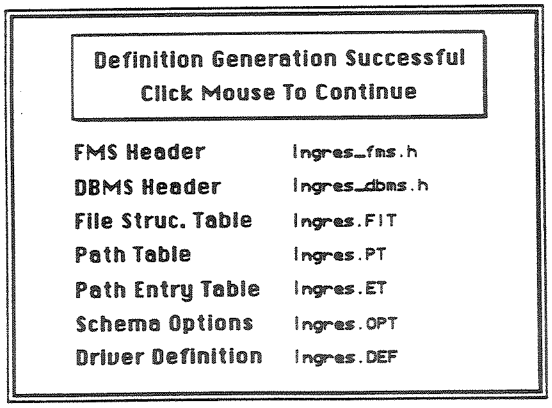
[6] **System Managers** and **System Utilities** refer to a standard package of ADTs (queries, into-lists, etc.) that are referenced by virtually all file mapping, link, and model components. **System Components** is a generic name given to components listed on the DBMS's operation bus. **File Manager** refers to FMS code that is generated. **File Layers**, **Link Layers**, and **Model Layers** refer to file mapping, link, and data model components that are referenced.



Note that the ease with which DBMSs with multiple DMLs can be created is a novel feature of Genesis. It is a capability that we intend to explore in future research. In particular, it is well-known that there is no standard for object-oriented DMLs; every OODBMS seems to have its own unique data language. We are now in the process of adding components to the Genesis library that will enable us to build OODBMSs as component assemblies, just as we are doing with relational DBMSs here. Multiple DMLs will give us the ability to experiment and evaluate different language syntaxes and features easily.

3.4.3 System Generation

A transcript of the design in an FMS or DBMS window is generated by pulling down the **File** menu and selecting **generate.** In the case of Ingres, the following screen is displayed:



Definition Generation Successful
Click Mouse To Continue

FMS Header	Ingres_fms.h
DBMS Header	Ingres_dbms.h
File Struc. Table	Ingres.FIT
Path Table	Ingres.PT
Path Entry Table	Ingres.ET
Schema Options	Ingres.OPT
Driver Definition	Ingres.DEF

The objects that are generated are configuration files. The **FMS Header** and **DBMS Header** files are C-preprocessor include files that are compiled with the Genesis library to produce **Ingres.** The **File Struc. Table, Path Table, Path Entry Table,** and **Schema Options Table** encode components interconnections and are read by DBMS executables to process dml operations. The **Driver Definition** is a DaTE readable document that is a copy of the DBMS window that defines Ingres. A similar screen is displayed for FMS generation.

Examining the contents of the configuration files is beyond the scope of this paper. This topic is considered in [Bat90].

3.4 Recap

Defining a target DBMS or FMS with DaTE takes about ten to fifteen minutes; the generation of configuration files takes seconds. To produce executables of the target system requires the compilation of the configuation files with the entire Genesis library. On a Mac IIcx using a Think C environment, this takes about fifteen minutes. On a SparcStation, compilation takes about twenty minutes. Details on the implementation of Genesis and the role of configuration files are discussed in [Bat90].

Beneath the graphical exterior of DaTE and hidden from DaTE users are sophisticated algorithms that ensure all DBMSs and FMSs that are designed are correct. How this is accomplished is the subject of the next section.

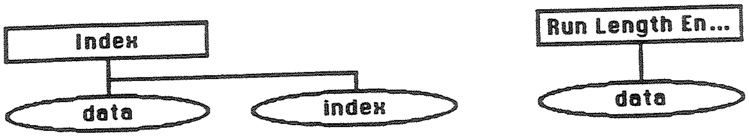
4. Design Rule Checking: The Validation of Compositions

Not all combinations of components are correct or meaningful. A major objective in our design of DaTE was to prevent DBMS implementors from making design errors; DaTE should permit only correct designs to be specified. Correctness is not the same execution speed; correctness means that the target system will always work.

The major questions that we faced were: (1) what are incorrect and meaningless DBMS designs? (2) What are design rules that prohibit the creation of such designs? And (3) what are efficient algorithms that enforce these rules? Solutions to these questions are presented in this section.

4.1 Examples of Incorrect Subsystems

As mentioned earlier, a component is akin to a parameterized type. Every parameter is identified with a realm of components or subsystems that can instantiate it. Consider the **data** parameter of the file mapping components **Index** and **RLE** (run-length encoding). It can be instantiated by a file mapping component or by a file structure. (That is, the concrete data files output by **Index** or **RLE** can either be transformed by a file mapping component, or can be stored directly in a file structure).



In the following discussions, it is important to remember that a compressed record, output by **RLE**, is just a string of bytes; values of the original fields can be reconstituted only by decompressing the entire record. This implies that uncompressed records may have primary keys, while compressed records do not. [7]

Consider the following compositions: (Fig. 4.1a) **Index** with **RLE** and (Fig. 4.1b) **RLE** with **Index**. Composition (4.1a) means that secondary indices over uncompressed fields are created, and then the data file is compressed. Composition (4.1b) is meaningless: once a file

[7] It could be argued that the entire compressed record is its own primary key. The problem here is that such a primary key is meaningless; B+ trees order records on primary keys, hash structures hash primary keys for storage, etc. All qualified searches of compressed files resort to scanning, which is equivalent to having no primary key at all.

is compressed, there are no fields to index. (Or rather, the fields that were designated for indexing are no longer present). Situation (4.1b) is avoided by a design rule that requires indexing to occur prior to compression.

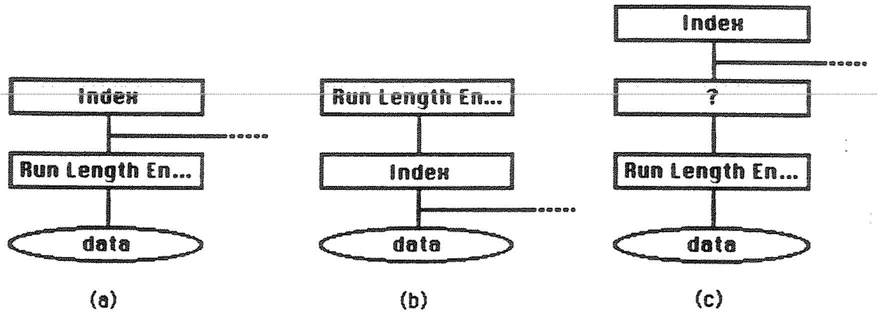


Figure 4.1 Compositions of Index and Run-Length Encoding Components

The difficulties of design-rule checking are amplified because offending components need not be adjacent in a composition. For example, the error of Figure 4.1b remains even if intermediate component(s) separate **RLE** and **Index** (see Fig. 4.1c)). In this example, the loss of a data file's primary key is permanent; intervening components cannot alter this fact.

Incorrect compositions of components with subsystems is another source of errors. Let **heap** and **bplus** be heap and B+ tree file structures. Figure 4.2a shows **RLE** composed with **heap** and Figure 4.2b shows **RLE** composed with **bplus**. Composition (4.2a) means that compressed files are stored in heaps. Composition (4.2b) is incorrect. Files must have a primary key in order to be stored in **bplus** trees; recall that compressed files have no such keys. A design rule for this situation is that the set of file structures that can instantiate the **RLE data** parameter must not require primary keys.

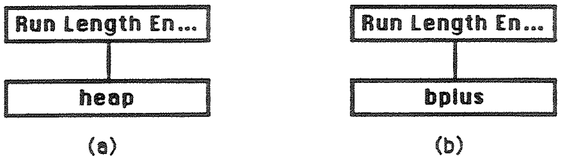


Figure 4.2 Compositions of Run-Length Encoding with File Structures

As a last example, consider the **Transposition** file mapping component which transposes a file into subfiles, and **seq_unord** which is an unordered file structure. (Unordered files differ from heaps in that internal pointers to records are record numbers (i.e., the *i*th record) whereas heaps return physical addresses). Transposition requires that the concatenation of the *i*th subrecord in each subfile reconstructs the *i*th record in the untransposed file. Composition (4.3a) is correct; each subfile is stored in an unordered file. Composition (4.3b) is incorrect. Corresponding subrecords in different subfiles will be assigned different internal pointers by heap algorithms. Composition (4.3c) is also incorrect, as subfiles don't have primary keys.

The above examples are representative of the errors that can arise when components are composed indescriminately. As we show in subsequent sections, error avoidance involves the enforcement of design rules.

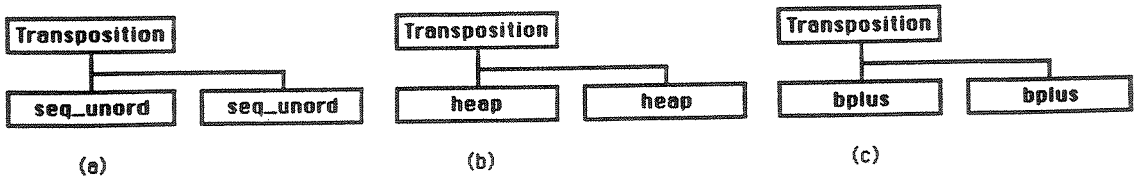


Figure 4.3 Compositions of Transposition with File Structures

4.2 A Model of Component Attributes and Restrictions

We use a model of boolean attributes and restrictions to eliminate DBMS design errors. An **attribute** is a boolean variable that specifies whether or not a component satisfies a given property. A **restriction** or **design rule** is a boolean variable that is associated with parameters of a component. It specifies the value an attribute must have if a component or subsystem can be used to instantiate a parameter.

Let ι be an attribute and A_ι denote its value. Every attribute ι has a corresponding pair of restrictions: R_ι and $R_{\bar{\iota}}$. $R_\iota = 1$ means that all components that can instantiate a parameter must have $A_\iota = 1$. Conversely, $R_{\bar{\iota}} = 1$ requires components to have attribute $A_\iota = 0$. The don't care or not relevant condition is the assignment $R_\iota = 0$ and $R_{\bar{\iota}} = 0$.

Each realm (or actually the components within the realm) is described by a set of attributes, called *realm attributes*. Every component, however, can impose restrictions on attributes from any realm. This means that higher components can dictate properties of components (potentially very far) below them. [8].

More specifically, let A be the boolean vector of *all* attributes and R be the vector of *all* restrictions. Every component L has a vector A^L, a subvector of A, which lists the values of the realm attributes of L. Every parameter P of L has a restriction vector $R^{L,P}$ which lists the restrictions imposed on components or subsystems that can instantiate P.

Examples. An attribute of file mapping components is {requires semantics} and attributes for file structures are {requires semantics, relative location key}. These attributes are defined below:

requires semantics Does component require field definitions and annotations of the file declared at conceptual level to be visible?

relative location key Are records assigned relative location keys (i.e., the *i*th record in a file) as internal identifiers by the component?

The values of these attributes for the components and subsystems referenced in Section 4.1 are:

[8] We have already seen an example of this. **Transposition** requires data files to be stored in unordered file structures (e.g., **seq_unord**). There can be any number of file mapping components that lie in between **Transposition** and **seq_unord**.

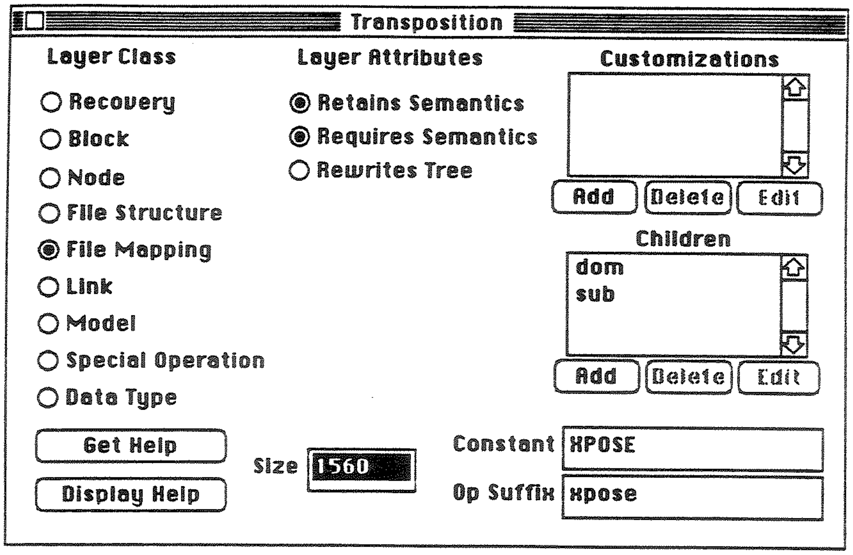
Attribute	Index	RLE	Transposition	bplus	heap	seq_unord
$A_{\text{requires semantics}}$	1	0	1	1	0	0
$A_{\text{relative location key}}$	*	*	*	0	0	1

* not applicable

The restrictions imposed on the parameters of the **Index**, **RLE**, and **Transposition** file mapping components are defined in the following table:

	LAYER-parameter				
Restriction	Index-data	Index-index	RLE-data	Transposition-dom	Transposition-sub
$R_{\text{requires semantics}}$	0	0	0	0	0
$R_{\overline{\text{requires semantics}}}$	0	0	1	1	1
$R_{\text{relative location key}}$	0	0	0	1	1
$R_{\overline{\text{relative location key}}}$	0	0	0	0	0

Attribute and restriction data, among other information, is entered at the time when a component is registered with DaTE. By choosing the **Component Def** option from the **File** menu, a window for component (layer) definition is displayed:



Transposition

Layer Class

○ Recovery
○ Block
○ Node
○ File Structure
● File Mapping
○ Link
○ Model
○ Special Operation
○ Data Type

Layer Attributes

● Retains Semantics
● Requires Semantics
○ Rewrites Tree

Customizations

Children

dom
sub

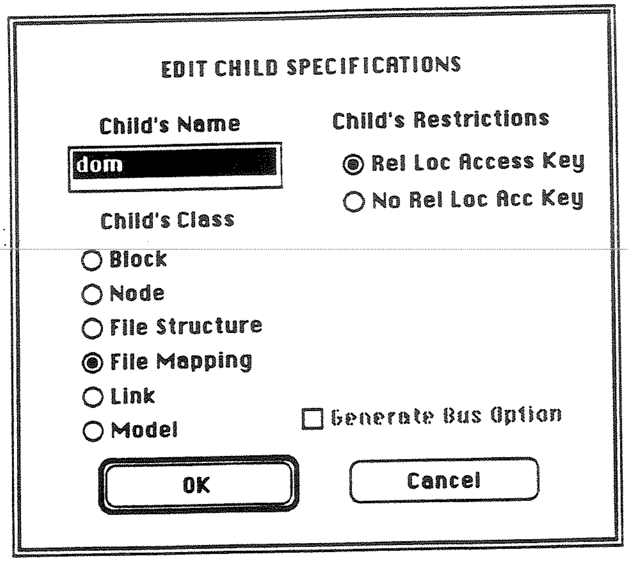
Size 1560

Constant XPOSE

Op Suffix xpose

The realm of the component is selected by clicking the appropriate button under **Component Realm** (Layer Class). Realm attributes are then displayed under **Component Attributes** (Layer Attributes):

A component can have any number of parameters. A parameter is entered by clicking the **Add** button under the **Children** scrollable window. Doing so causes a child specification window to appear:



EDIT CHILD SPECIFICATIONS

Child's Name: dom

Child's Class:
○ Block
○ Node
○ File Structure
● File Mapping
○ Link
○ Model

Child's Restrictions:
● Rel Loc Access Key
○ No Rel Loc Acc Key

□ Generate Bus Option

The realm of the child parameter must be specified, along with the restrictions that are imposed on that parameter.

The current version of DaTE deals with eight attributes.[9] The maximum number of attributes per realm is three; the maximum number of restrictions is eight.

The total number of attributes (and restrictions) is open-ended. In principle, each time a new component is registered with DaTE, there may be additional attributes and restrictions that need to be introduced in order to avoid certain combinations of components. Such changes, at present, must be hard-coded into DaTE, and this can only be accomplished by a sophisticated user. In choosing the current set of attributes, we have surveyed a large spectrum of components (much greater than the current number that are presently available) to minimize the likelyhood that DaTE will need to be altered.

As an aside, other information that is collected at component definition time is a help picture, the size of the component in lines of C code, labels for C preprocessor constant generation, and a list of customizations.

4.3 Design Rule Checking Algorithms

The legality of compositions is checked at the time when parameters are instantiated. When a parameter is clicked, DaTE displays a library window containing only those components or subsystems that satisfy the restrictions imposed on that parameter. The algorithms used to determine legal components and subsystems are explained below.

The graphics of DaTE reflect the abstract-to-concrete mapping of files; restrictions are inherited vertically, from top downwards. Let $P_0 \cdots$ be the sequence of parameters whose instantiation defines a vertical (conceptual-to-dominant-internal) path of components. Let L_j be the component that instantiates P_j. The restrictions $F_{\iota,j}$ and $F_{\bar{\iota},j}$ that are imposed on component L_j are the boolean disjunctions of the restrictions on A_ι imposed by each of L_j's ancestors:

[9] Technically, there are 8+n attributes, where n is the number of file mapping components. No file mapping component can appear twice in a vertical column of an subsystem (i.e., abstract-to-concrete mapping of the dominant file). Thus, **RLE** composed with **RLE** or **Index** composed with **Index** are avoided. Generally, compositions of the same component with itself are not meaningful. A technical reason, concerning the implementation of Genesis, also forces this restriction.

$$F_{\iota,j} = \bigvee_{k=0}^{j-1} R_{\iota}^{L_k,P_k}$$

$$F_{\bar{\iota},j} = \bigvee_{k=0}^{j-1} R_{\bar{\iota}}^{L_k,P_k}$$

That is, once a restriction is imposed by an ancestor component, it cannot be removed.

To simplify notation, let F_{ι} and $F_{\bar{\iota}}$ denote the restrictions on attribute ι that a component or subsystem must satisfy. Let $M_{\iota} = 1$ if a specific value is required for A_{ι}, 0 otherwise. Let the specific value be V_{ι}. The following identities are evident:

$$M_{\iota} = F_{\iota} \vee F_{\bar{\iota}}$$

$$V_{\iota} = F_{\iota}$$

It is possible that restrictions can be imposed on attributes that are not among the realm attributes of the component to be selected. For example, requiring that relative location keys be used is a restriction on the choice of legal file structures, not file mapping components. We are interested in examining only those realm attributes for which restrictions are imposed. Let K be a realm of components, and let $C_{\iota} = 1$ if ι belongs to K's realm attributes, 0 otherwise. Let E_{ι} be the boolean variable that indicates whether or not A_{ι} should be examined:

$$E_{\iota} = C_{\iota} \wedge M_{\iota}$$

Let L be a component of realm K, and A_{ι}^{L} be the value of ι for L. L satisfies the restrictions on ι iff Qual(L,ι) is true:

$$\text{Qual}(L,\iota) = (\neg E_{\iota}) \vee (E_{\iota} \wedge (V_{\iota} = A_{\iota}^{L}))$$

It follows that L is qualified to instantiate a parameter iff it is qualified on all realm attributes:

$$\text{Qualified}(L) = \bigwedge_{\iota \text{ in set of all attributes}} \text{Qual}(L,\iota)$$

The components or subsystems of a realm that satisfy the Qualified( ) function are precisely those that can instantiate the selected parameter without violating design rules. These components or subsystems are the building blocks that are listed in library windows displayed by DaTE. By limiting the selections to legal choices, incorrect subsystems cannot be specified.

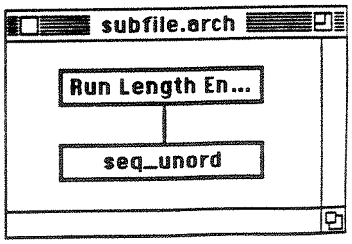
The algorithm for evaluating Qualified( ) follows directly from the above definition. The algorithm has O(n*m) complexity, where n is the path length and m is the number of attributes. An improvement is to use integers to encode (short) boolean vectors, so that vector manipulation complexity is approximately O(1) instead of O(m). Further improvements are attained by remembering the restrictions that have accumulated after each instantiation, so that only the parent of a component, not all ancestors of a component, needs to be examined. With these improvements, the complexity of design rule checking is approximately O(1).

It is worth noting that it is possible for file structures and component definitions to be modified once they are created. DaTE currently does not revalidate existing compositions when such modifications occur. We are presently exploring ways to solve this problem. The simplest solution is to disallow updates to subsystems once they are complete and have been saved. Anything more sophisticated requires version control.

3.4 Attributes of Subsystems

In our design rule checking algorithms, attribute values for components and subsystems were assumed to be readily available. Values for components are specified manually; for subsystems, they are computed. How values are computed is the subject of this section. Since subsystems are treated as primitive and nonparameterized components in DaTE, subsystems impose no restrictions.

Recall the storage subsystem **subfile.arch** for subordinate files of transposed files. Each subfile is run-length compressed before being stored in an unordered file. The attributes of this subsystem are $A_{\text{requires semantics}} = 0$ and $A_{\text{relative location key}} = 1$. (That is, **subfile.arch** does not require records to have their original field semantics and it assigns relative location keys to records).



Because this subsystem encapsulates both file mapping and file structure concepts, it must exhibit attributes of both. The attribute $A_{\text{requires semantics}} = 0$ is inherited directly from the **RLE** component. (Reason: no matter what components lie beneath **RLE**, the value of this attribute cannot change). $A_{\text{relative location key}} = 1$ is inherited from the **seq_unord** file structure subsystem.

In general, the realm attributes of an subsystem are the union of the realm attributes of each component along its dominant (left-most vertical) path. The values of these attributes are inherited from these components. A complication that arises is that some components along a path share the same attributes and possibly assign them different values. In such cases, priority is given to the component that is closest to the subsystem's root. That is, if attribute ι is shared by many components, and L is the component closest to the root that has ι, then A_ι is taken from L. The justification for this priority is encapsulation; once an attribute (and its value) is exposed, it becomes part of the subsystem's interface. An implementation of this interface may ultimately alter this value at lower components, but encapsulation means that such changes are invisible.

Subsystem_Attributes(Arch) is the algorithm used in DaTE for computing the attribute values of a complete subsystem Arch that does not have software busses. (Extensions for software busses are considered in the next section). In the following, let L be a component or subsystem. Let dominant(L) return the root node of L (if L is an subsystem), the dominant child node of L (if L is a component and if children exist), or null (otherwise). Let attributes(L) return the set of realm attributes of L.

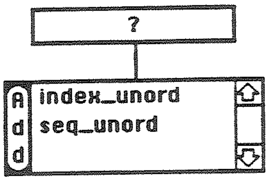
```
Subsystem_Attributes( Arch )
    {initialize vector of subsystem attributes}  foreach $\iota$  { $A_\iota^{Arch} = 0$ }.
    {initialize attribute set to empty set}  $AS = \varnothing$.
    {get root node}  L = dominant( Arch ).
    {loop on nonnull node}  while (L != null) {
        {extra is set of attributes whose values are to be assigned}
            extra = set_difference(attributes(L),AS).
        {assign attribute values}  foreach $\iota$ in extra  { $A_\iota^{Arch} = A_\iota^{L}$ }.
        {add extra attributes to AS}  AS = union(AS, extra).
        {get node of dominant child}  L = dominant(L).
        }
    {return attribute vector as result}  return( $A^{Arch}$ ).
```

DaTE executes Subsystem_Attributes( ) at the time an subsystem is saved.

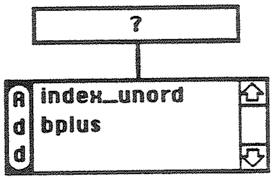
3.5 Handling Software Busses

A software bus lists two or more components or subsystems that satisfy the same restrictions and belong to the same realm. Bus members may have any number of attributes on which no restrictions are placed. It is quite common for different members to assign different values to unrestricted attributes. Since a software bus acts as a single primitive component that has the same realm attributes as its members, the question arises how does one assign values to its nonrestricted attributes?

The solution we have adopted is to take the boolean conjunction of the realm attributes of each object on a bus. Our reasoning is that if a bus has $A_i = 1$, then all members of the bus must exhibit $A_i = 1$. This provides a practical solution to the following problem. Two storage subsystem are shown below. System_A stores concrete files in different types of unordered file structures. System_B uses unordered and B+ tree file structures. System_A could be used to store subordinate subfiles of transposed files, as any of its file structures satisfies the relative location key requirement. System_B cannot be used, since B+ trees might be selected for file storage, which would be wrong.



System_A



System_B

Note that we could have chosen the interpretation of setting $A_i = 1$ if *any* member of a bus has $A_i = 1$. This would lead to incorrect designs, as the above example illustrates. However, it would seem possible that an subsystem with $A_i = 0$ might have some - not all - components/subsystems with $A_i = 1$. Thus, incorrect subsystems might arise when the *absense* of an attribute is required (i.e., $R_{\bar{i}} = 1$).

We have circumvented this difficulty in DaTE with a provisional solution. DaTE insists that (a) the root of every storage system subsystem is a single component, not a software bus, and (b) requirements imposed on inherited attributes of nonroot components are affirmative (i.e., $R_i = 1$). Condition (a) implies that the top component of a storage system determines almost all attribute values of the subsystem. The only attribute (currently) whose value remains to be assigned is relative_location_key. The restrictions that can be imposed on this attribute are $R_{relative\_location\_key} = 0$ or 1 and $R_{\overline{relative\_location\_key}} = 0$, which satisfies condition (b). Removing these limitations will require a more general model of attributes and restrictions than we are presently using.

4. Conclusions

Software systems are becoming progressively more complex and difficult to construct. Building-block technologies will become increasingly important as the reinvention and recoding of known technologies becomes economically unattractive. Genesis is one of the first software building block technologies that embodies large-scale reuse; it is a proof-of-concept system. DaTE, our software layout editor, enables DBMS implementors to design entire database systems or portions thereof that are customized for a target application. (For example, DaTE could help generate a relational engine that is a component of object-oriented database system). If all software components are available, system assembly can take minutes and yield enormous increases in software productivity.

We have explained some of the features of DaTE. Its expressibility is limited primarily by the components that are available. Extending DaTE to capture object-oriented DBMSs, concurrency control and recovery for multi-client DBMSs, distributed DBMSs, and large-grain parallelism that is inherent in internal DBMS algorithms is indeed possible [Bat88]; however, more work needs to be done.

We also presented the mechanism we are currently using for design rule checking. We have explained how DaTE makes it impossible to specify DBMS and FMS designs that cannot possibly work. Additional research is needed to generalize our design rule checking algorithms and attribute-restriction model in the handling of software busses.

References

[Bat85] D.S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', **ACM Trans. Database Syst.**, 10,4 (Dec. 1985), 463-528.

[Bat88a] D.S. Batory, 'Concepts for a Database System Synthesizer', **ACM PODS 1988.**

[Bat88b] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, 'GENESIS: An Extensible Database Management System', **IEEE Trans. Software Engr.**, 1711-1730.

[Bat89a] D.S. Batory, J.R. Barnett, J. Roy, B.C. Twichell, and J. Garza, 'Construction of File Management Systems From Software Components', **COMPSAC 1989.**

[Bat89b] D.S. Batory, 'On the Reusability of Query Optimization Algorithms', **Information Systems**, 1989.

[Bat90] D.S. Batory, et. al, 'The Design and Implementation of Genesis 2.0', 1990.

[Bat91] D.S. Batory and S.W. O'Malley, 'On the Design and Implementation of Hierarchical Systems from Reusable Components', 1991.

[Big89] T.J. Biggerstaff and A.J. Perlis, *Software Reusability*, ACM Press, 1989.

[Car88] M. Carey, et al., 'A Data Model and Query Language for EXODUS', **ACM SIGMOD**, 1988.

[Haa89] L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh, 'Extensible Query Processing in Starburst', **ACM SIGMOD**, 1989.

[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.

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

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

Department of Computer Sciences
The University of Texas at Austin

The Genesis Database System Compiler: User Manual

Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

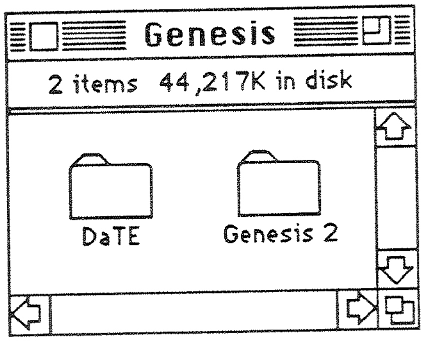
August 1990

Introduction

Genesis is the first software building-blocks technology for database management systems. It is also one of the first examples of large scale software reuse. This paper describes how to assemble DBMSs using Genesis. The design and implementation techniques utilized in Genesis are described elsewhere. Instructions on how to install Genesis on a Mac II are given in the Appendix along with a list of known bugs.

Getting Started

Genesis consists of a configuration editor (DaTE) and prewritten software modules called **layers**. On Macintoshes, the **Genesis** folder contains all Genesis software. The **DaTE** folder contains DaTE and its files; the **Genesis 2** folder contains building-block source code:

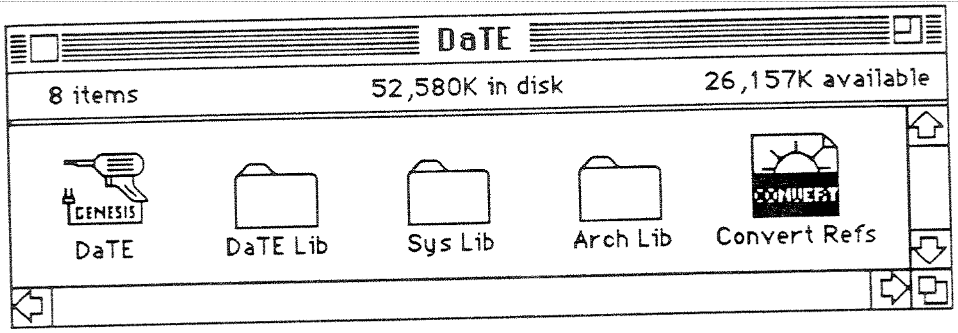


There are three phases in the life-cycle of a Genesis DBMS: specification, assembly, and usage. The **specification phase** involves using DaTE to define a target database management system. The output of DaTE is a set of **configuration files** that specify the interconnections between layers of the target system. The **assembly phase** is the actual creation of DBMS executables. This is accomplished by compiling the generated configuration files with the Genesis library. The **usage phase** deals with assembly validation, schema creation, compilation, database loading, and database processing.

The following chapters will explain each of these phases in more detail. As a running example, we will show how an approximation to University Ingres can be generated.

Phase I: DBMS Specification

The most complicated and most interesting phase of DBMS generation is that of specification. **DaTE** (or Database system Type Editor) is a graphical language for composing software building-blocks. The DaTE folder contains the DaTE and Convert Refs applications and three folders: DaTE Lib, Arch Lib, and Sys Lib. **DaTE Lib** has definitions of all primitive Genesis building blocks. **Arch Lib** and **Sys Lib** respectively contain the architectures and systems generated by DaTE.



Clicking DaTE begins its execution, which starts by reading primitives from the DaTE library. **Convert Refs** is needed only to transport architecture designs from one disk to the next. We'll consider its use at the end of this Phase.

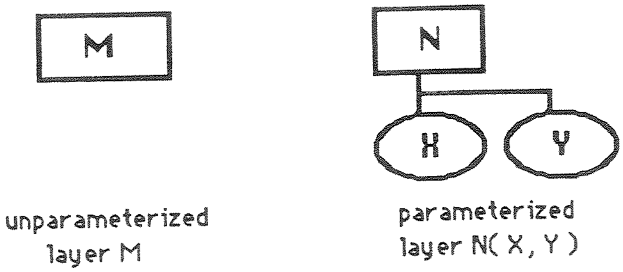
(Note: as of this writing, the **Sys Lib** and **Arch Lib** folders are a convention which we strongly recommend readers to follow; they are not required by DaTE. The **DaTE Lib** folder is required).

Historical note: Primitives in the DaTE/Genesis library were considered, at one time or another, to be software ICs. The Genesis icon - a wire wrapper or soldering iron - was chosen to symbolize the interconnection of ICs into software systems. The permanence of this icon remains to be seen.

The name DaTE - Database system Type Editor - was chosen at a time when the distinction between layers and parameterized types was not recognized. The permanence of "DaTE" as a name also remains to be seen.

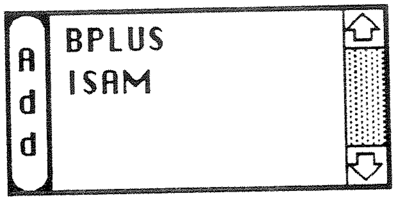
DaTE Diagrams

Building-blocks of Genesis are parameterized **layers**, a concept akin to a parameterized type. DaTE depicts layers as boxes and parameters as ovals. Layers M and N are shown below; M is unparameterized and N has two parameters, X and Y:



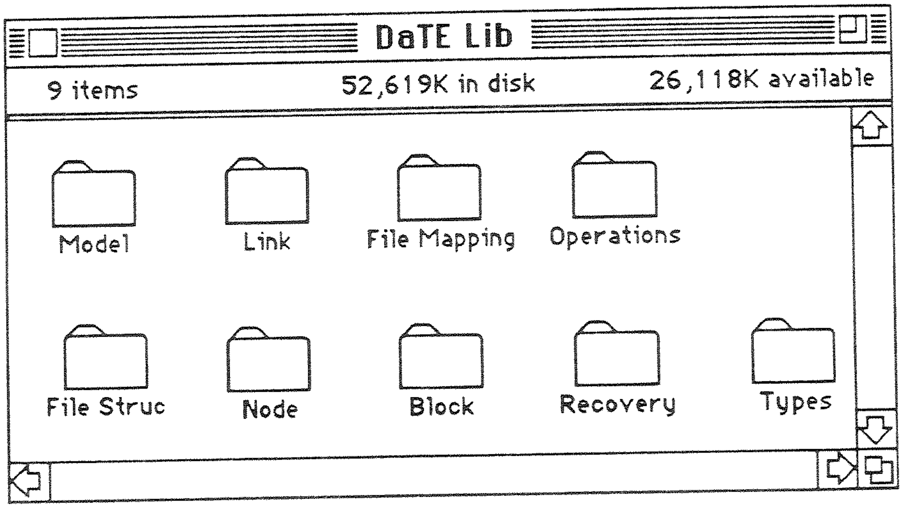
An **architecture** is a rooted graph. DaTE supports the definition of four progressively more complex types of architectures: file structure, storage system, network, and relational. An architecture is **complete** if it has no unbound parameters (i.e., no ovals). Complete architectures are treated as primitive, unparameterized layers by DaTE.

A fundamental concept in DaTE is the **software bus**. It is an abstract construct that allows multiple layers to occupy the same position in an architecture. Software busses are depicted as a scrollable window. The bus below lists the layers (or architectures) BPLUS and ISAM:



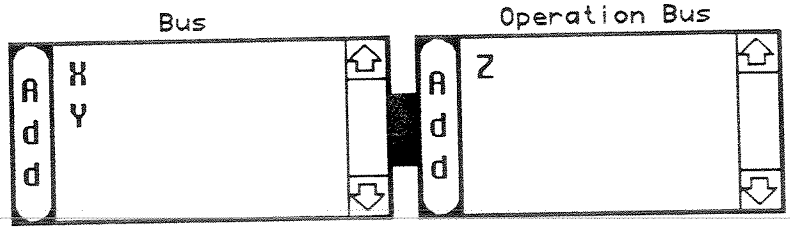
The **Add** button admits new entries to a bus. An entry is deleted by clicking it and choosing the **Remove** option from the displayed popup menu.

Layers and architectures belong to classes. There are the classes of file mapping layers, link layers, etc., as well as the class of file structure architectures, the class of storage system architectures, and so on. If you open the DaTE Lib folder, you'll see the classes presently available in Genesis. **File Struc**, for example, is a folder containing different file structure layers. **Recovery** is a folder containing different page-based recovery layers.



Getting back to software busses, DaTE restricts entries of a software bus to belong to a single class. Thus, all entries of a bus are storage systems, or all are link layers, etc. In principle, it may be possible (and useful) to have polymorphic software busses; this is a concept that needs to be explored in the future.

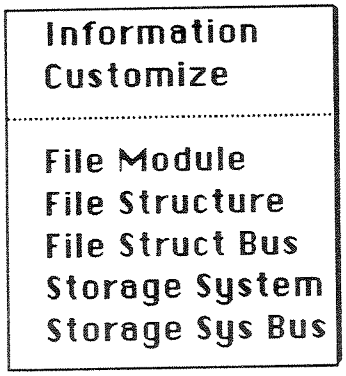
A bus can be extended by an **operation bus**, which permits the entries of a bus to reference special operations. An operation bus is attached to the software bus it is to complement:



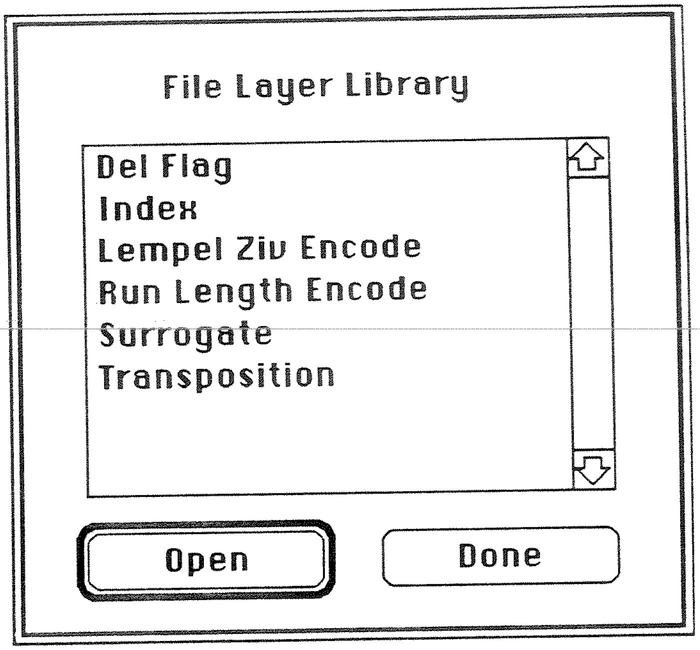
The semantics are straightforward: any entry in the primary bus may reference any entry in the operation bus. The above figure shows entries X and Y in the primary bus, and they may reference entry Z of the operation bus. Operation busses arise only in FMS and DBMS specifications.

Instantiating Parameters and General Editing Rules

Parameter instantiation in DaTE is accomplished by clicking an oval. The standard response of DaTE is to display a menu, like the one below:



Selecting **Information** displays a help window on the selected item. **Customize** lists the customizable options of a layer, and allows options to be enabled or disabled. Selecting an entry below the dotted line causes a scrollable **library window** to be displayed. The members of the library are legal layers or architectures that can instantiate the selected parameter. A library window for File Layers is shown on the top of the next page.



Occasionally, parameters are bound incorrectly (or better choices are later discovered). Rebinding a parameter is accomplished by clicking the box that is presently bound to the parameter. Choosing an alternative binding causes the previous selection to be overridden. Because DaTE imposes a top-down design methodology, all hierarchical bindings of the earlier layer may need to be erased as they might no longer apply to the new module. DaTE tries to save such bindings whenever possible.

Architectures and Systems

As mentioned earlier, an architecture is a rooted graph of primitive layers. There are file structure, storage system, network, and relational architectures. A **System** is a composition of one or more architectures and supporting primitive layers, such as data types, recovery, and special operations. File Management Systems (FMSs) and Database Management Systems (DBMSs) can be defined by DaTE. Configuration files, which are used in Phase 2 to assemble target systems, can be generated for FMSs and DBMSs.

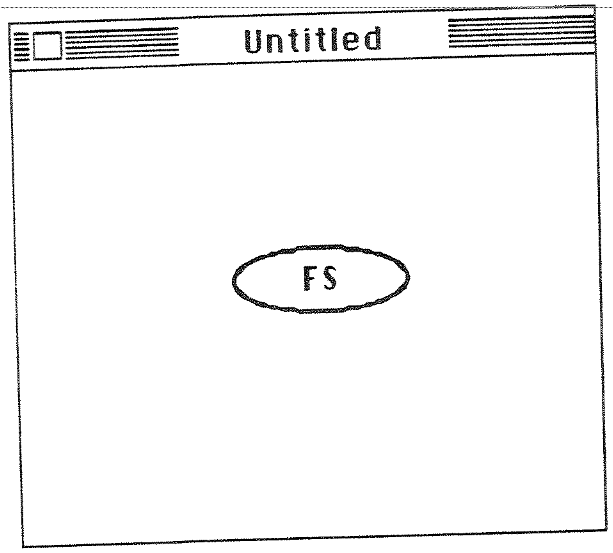
In the following sections, we explain how architectures and system are specified, starting with simplest and progressing to the most complicated.

Note: Each architecture and each system is stored by DaTE in its own file. Architectures, systems, and configuration files quickly become numerous, and placing them in a single folder is not a good idea. We strongly recommend that architectures be stored in the **Arch Lib** folder and systems and their configuration files be stored in the **Sys Lib** folder provided with DaTE.

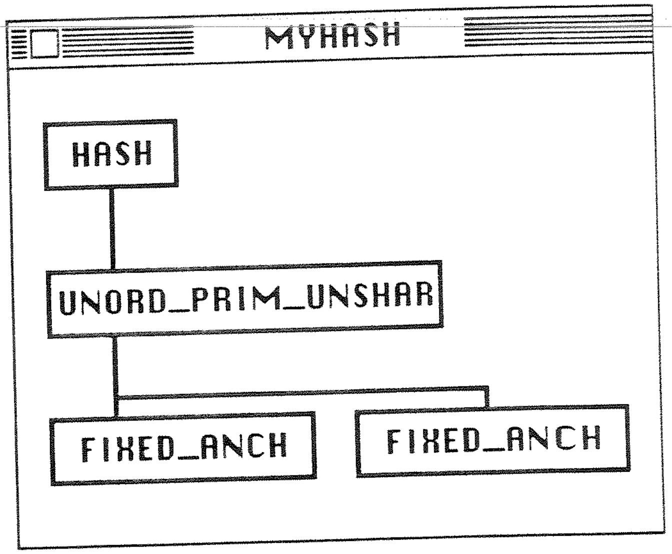
Creating a File Structure Architecture

A file structure is a composition of layers that provide the most primitive file storage and retrieval capabilities needed for DBMS operation. Genesis decomposes file structures into three distinct layers: FS (file storage), logical block (or nodes), and physical block.

A file structure is created in DaTE by pulling down the **File** menu, selecting **New**, and then **File Structure**. An empty window is then displayed (a):



(a)



(b)

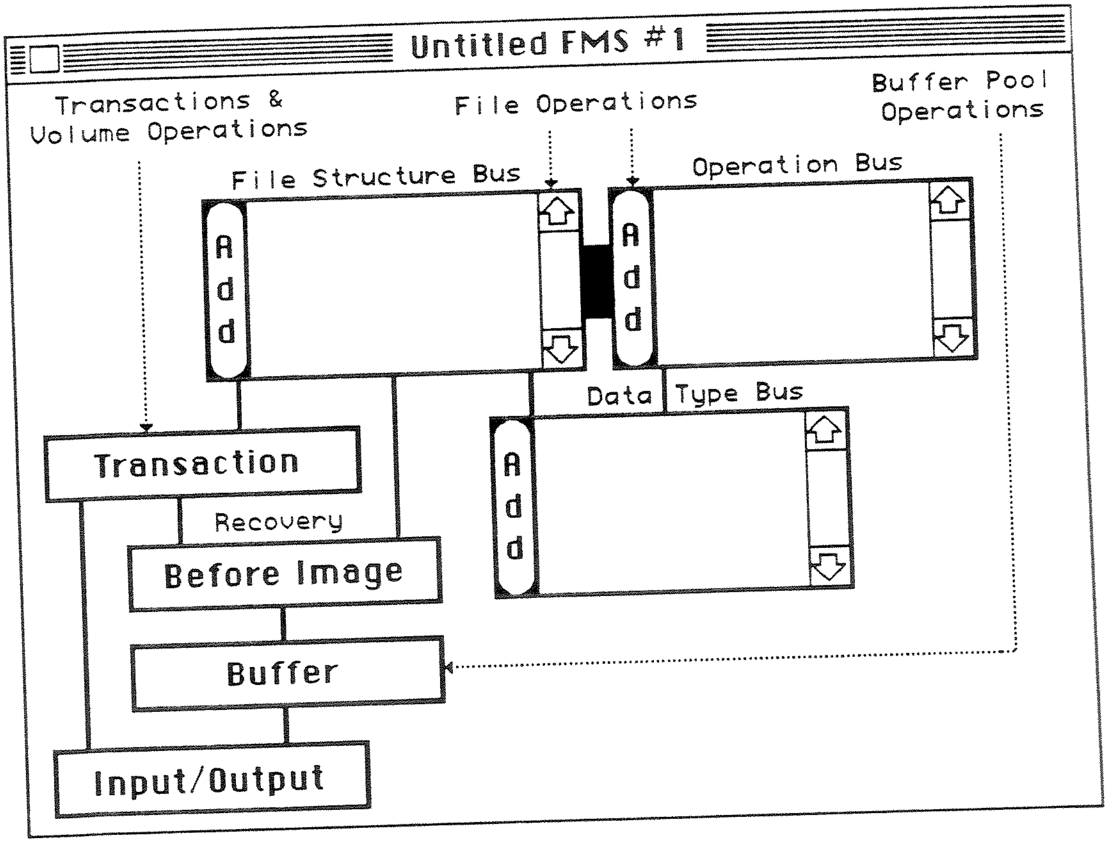
The window contains a single FS oval, indicating that a file storage layer must be specified. Clicking the oval causes a library window to appear that lists all FS layers known to Genesis. Clicking a layer selects it, and its box is displayed in the window. Depending on the layer, one or more Node ovals will hang from the layer box. Clicking these ovals in the same way allows Node implementations to be selected. Nodes have Blocks as parameters, thereby producing a tree of three levels, as shown above in (b). This particular window shows a HASH file structure whose data nodes are implemented by the UNORD_PRIM_UNSHAR layer (i.e., unordered records stored in a primary block with unshared overflow), and primary and overflow physical blocks are implemented by the FIXED_ANCH layer (i.e., fixed-length records with anchored physical addresses).

For a file structure architecture to be referenced later, it must be named and saved. This is accomplished by pulling down the **File** menu and selecting **Save** or **Save As....** The above architecture was saved with the name 'MYHASH'.

Creating a FMS

A **file management system (FMS)** is the kernel of a DBMS. It provides basic access methods, buffer management, and recovery capabilities necessary for DBMS operation. Genesis provides a standardized architecture for FMSs, where the design customization decisions have been factored into the selection of file structures, special operations, data types, and a recovery layer.

An FMS is created in DaTE by pulling down the **File** menu, selecting **New**, and then **FMS**. An empty FMS window is then displayed:

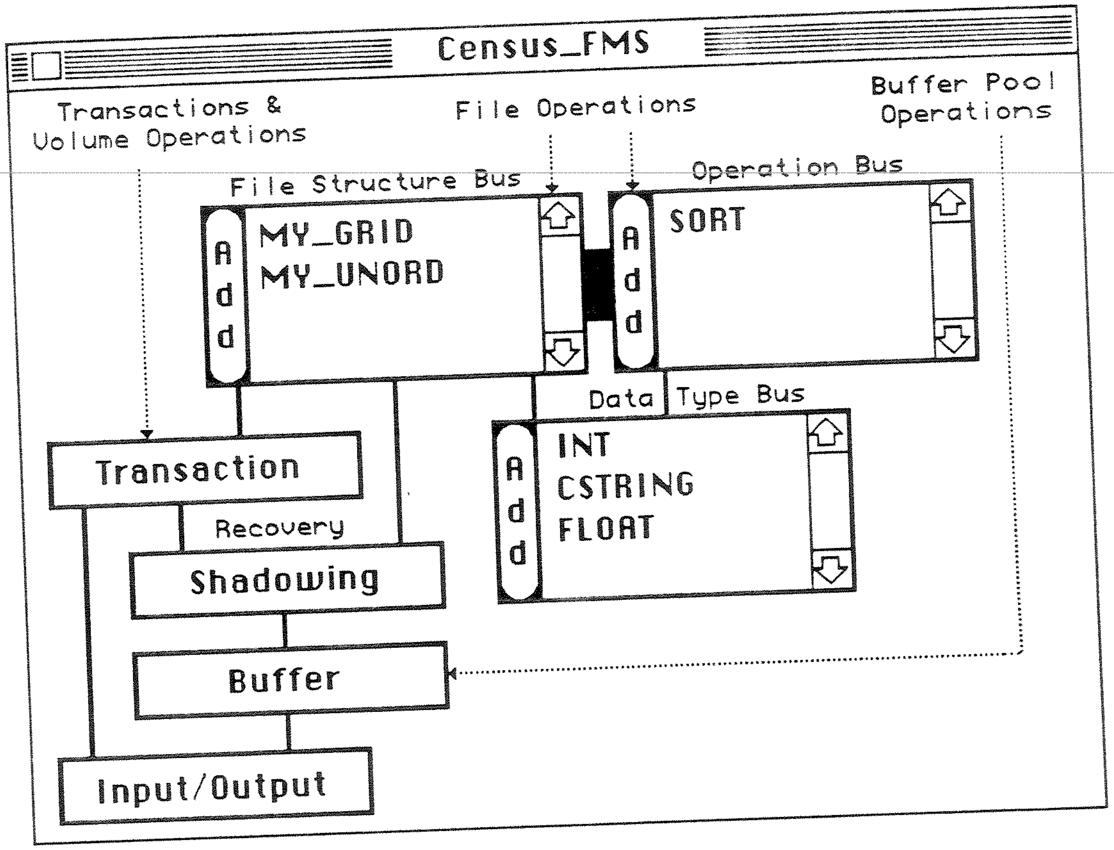


The above mentioned design decisions are entered in three busses and a field. Each labeled in small font as a prompt to an FMS designer. **Before Image** (Page) logging is assumed as the default implementation of recovery.

Note: **Transaction**, **Buffer**, and **Input/Output** classes in an FMS architecture are not customizable in this version of DaTE, as Genesis provides only a single implementation (i.e., layer) for each. When multiple layers are available, they too will be customizable.

Also note that an FMS window documents the routing (via dotted lines) of user-issued operations. Volume and transaction operations are serviced by the Transaction layer; buffer pool operations are handled by the Buffer layer. File operations are processed by either file structures or special operations. Observe that an FMS is not a strict hierarchy of layers, where all operations are transformed by layers in a top-down manner.

A possible FMS for a census database is shown below. It provides MY_GRID and MY_UNORD as primitive file structures. The SORT operation is included, along with the data types INT, CSTRING, and FLOAT. Shadowing is used for volume recovery.



To generate the configuration file of an FMS, pull down the **File** menu and select **generate**.

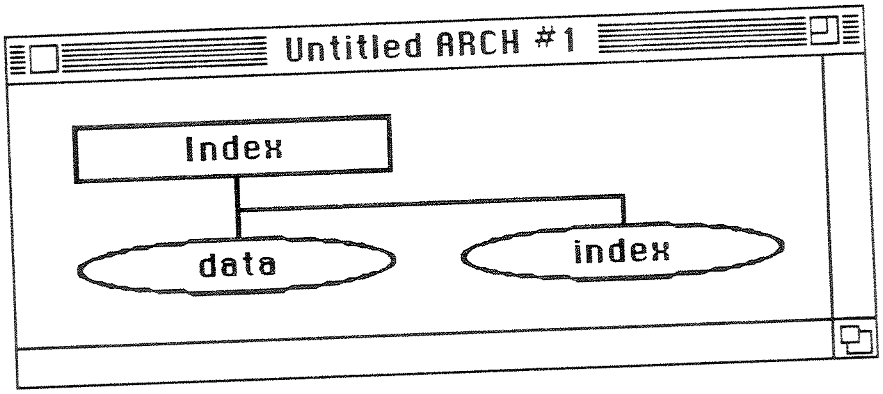
Note: Again, we recommend that FMSs and their configuration files be placed in the **Sys Lib** folder, so that they are separated from their architecture components. The files that are generated are text files that contain C precompiler directives and macro definitions. As we'll see in a later section, these are the files that are compiled with Genesis source to produce DBMS executables.

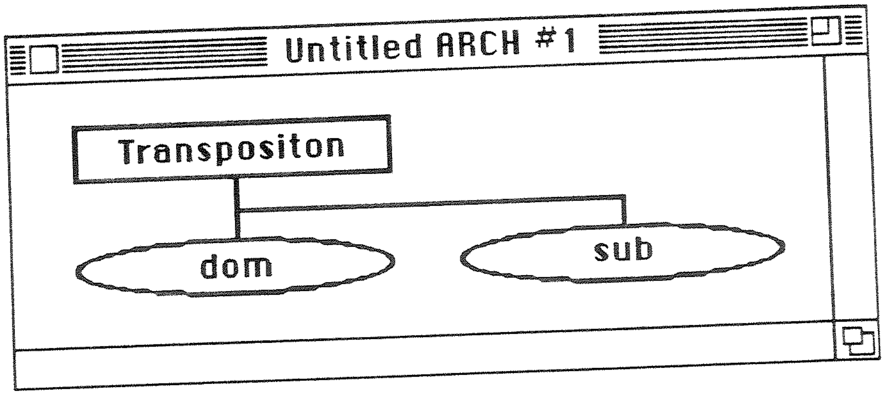
Note: As a general rule, one probably doesn't want to generate only an FMS. When a DBMS is generated, its underlying FMS is also generated. In fact, generating an FMS produces a _fms.h file, while generating a DBMS produces both a _fms.h file and a g_dbms.h file. For now, it is instructive to see what is going on internally with DaTE, although the generation of an FMS is likely to be a rare event for most Genesis users.

Creating a Storage System Architecture

A **file mapping layer** maps an abstract file to one or more concrete files. Examples include mapping a file to an inverted file (i.e., indexing), and mapping an uncompressed file to a compressed file (i.e., compression). Parameters of a file mapping layer are implementations of the concrete files that it generates. A **storage system architecture** is a composition of file mapping layers that terminate with file structure architectures.

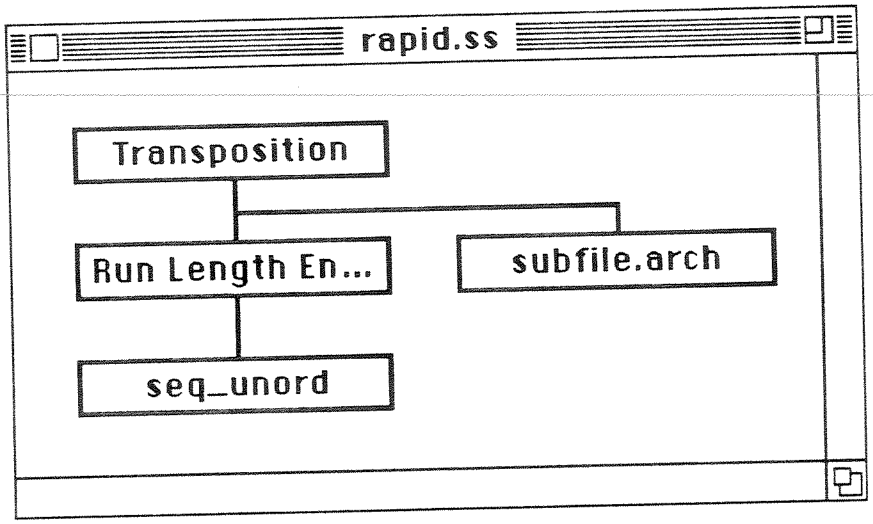
A storage system is created in DaTE by pulling down the **File** menu, selecting **New**, and then **Storage System**. An empty storage system window is then displayed. As examples of file mapping modules, the following two windows show the selection of indexing and transposition layers. **Index** maps an abstract file to a data file and zero or more index files. The data file implementation is specified by parameter **data** and the index file implementation by parameter **index**. Similarly, **Transposition** maps a file to a series of concrete subfiles, one dominant subfile and zero or more subordinate subfiles. Their implementations are specified via parameters **dom** and **sub**.

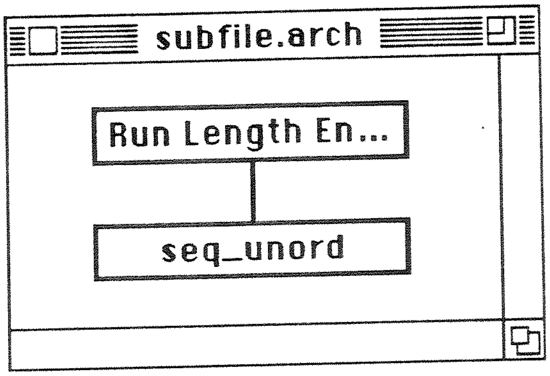




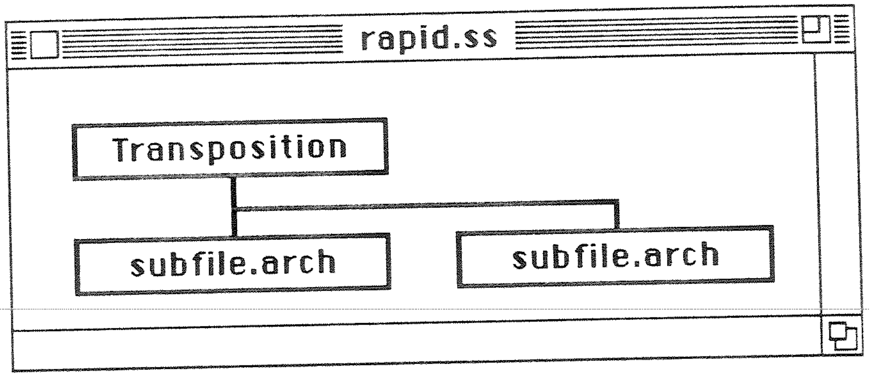
Note: The concept of dominance is fundamental to conceptual-to-internal mappings. Basically the idea is that a conceptual file is mapped by DBMS software to multiple internal files. One of the internal files is distinguishable as its records are in 1-to-1 correspondence with conceptual tuples. This is the **dominant** file. Other internal files, called **subordinate**, do not have this property.

Consider the storage system used by Rapid, a statistical DBMS. Rapid mapped schema-defined files to transposed files, where each column was run-length compressed before being stored in a sequential-unordered file structure. This storage system is defined in two storage system windows: **rapid.ss** and **subfile.arch**. Rapid.ss maps a schema-defined file to its dominant internal counterpart. Subfile.arch maps subordinate subfiles to their internal counterparts.



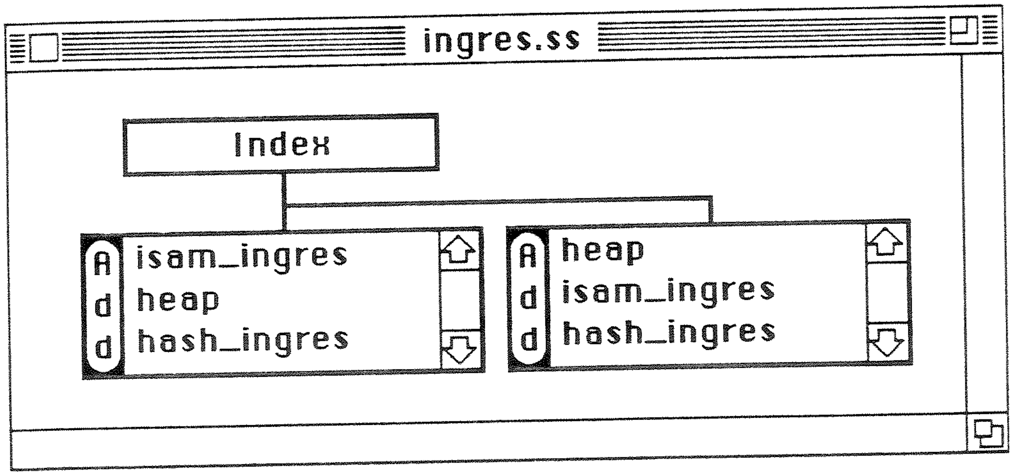


While it seems odd not to have Transposition call subfile.arch twice (as implementations of both dominant and subordinate files are the same), DaTE permits only one architecture reference per dominant mapping. (It turns out that permitting multiple storage system references significantly increases DaTE's complexity without providing greater expressibility. Otherwise, there is no apriori reason why it cannot be handled).



A composition that cannot be defined directly in DaTE....

As another example, consider the storage system of Ingres: it maps schema-defined files to inverted files, where data files and index files can be selectively implemented by hash, heap, or isam structures. The multiplicity of implementation choices is captured by a pair of file structure busses.



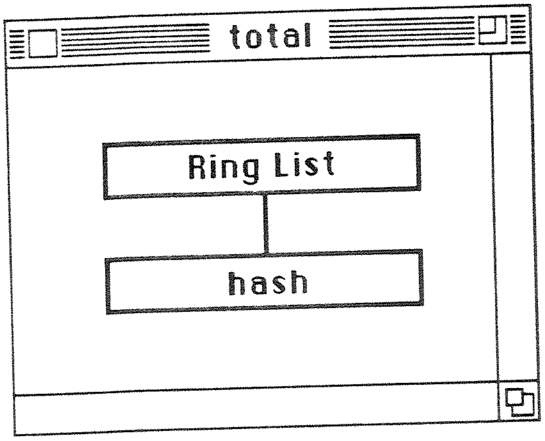
Note: The last entry on a software bus is the default mapping. Thus, data files in ingres.ss default to hash-based structures if no storage structure directive is provided. (These directives are specified in schemas, which is discussed in Phase 3). No other significance is attributed to the ordering of entries on a bus.

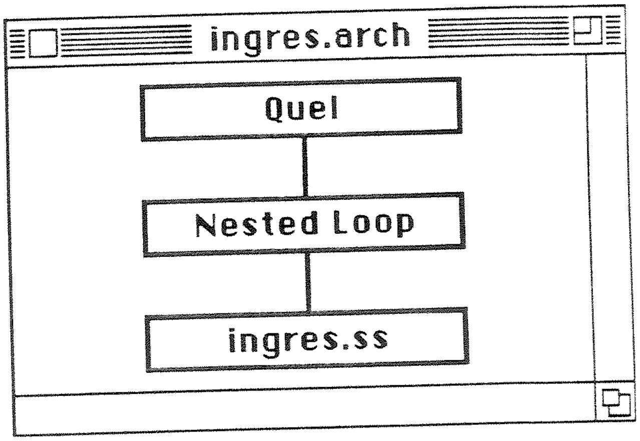
Creating a Network and Relational Architecture

A **network architecture** is rooted by a link layer (or link bus). This layer (or bus) specifies how links - i.e., relationships between files - are to be implemented. The sole parameter of a link layer (or bus) is the implementation of the referenced files, which may be expressed as a file structure or storage system architecture.

A **relational architecture** is rooted by a data model layer (or data model bus). Such layers map nonprocedural data model/data language interface to a procedural network database interface. The sole parameter of a data model layer is the implementation of the links of the network database.

The windows at the top of the next page show a network architecture used in the Total DBMS (i.e., no high-level data model; links are implemented by ring-lists,and files are stored in hash-based structures), and the relational architecture of Ingres (i.e., QUEL as the data model/language, nested loop implementations of links, and files stored in the Ingres storage system).



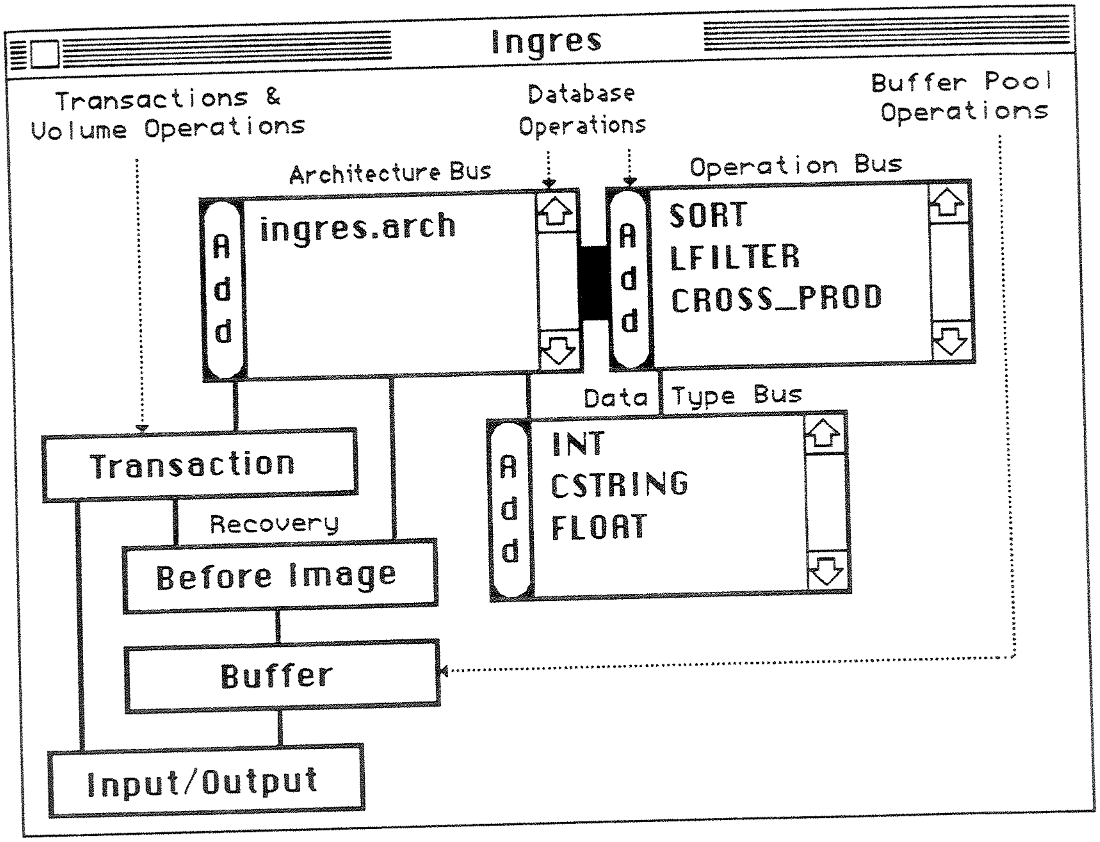


Creating a DBMS

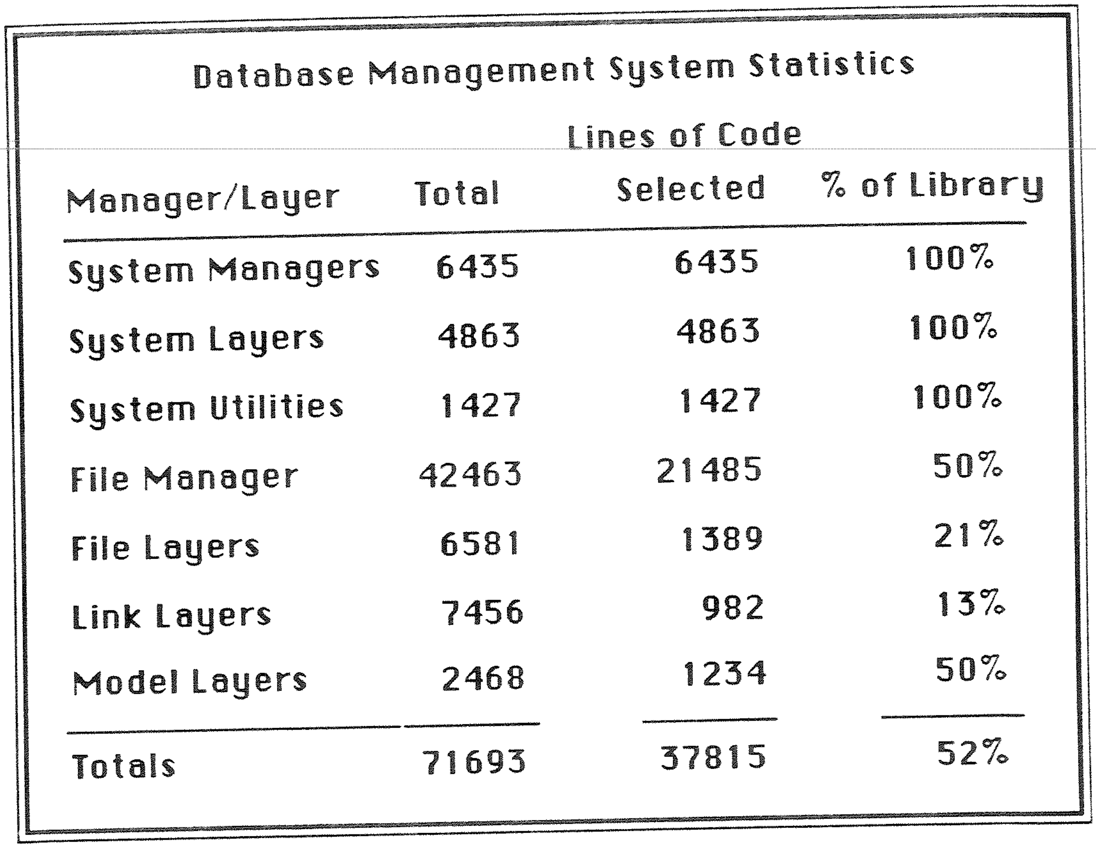
A DBMS is defined in a manner identical to that of FMSs: architectures, special operations, data types, and a recovery layer must be specified. The only significant difference is that relational, network, and storage system architectures are referenced instead of file structure architectures.

A DBMS window for our approximation of Ingres is shown below: the architecture is Ingres.arch; special operations are SORT, LFILTER, and CROSS_PROD; data types are INT, CSTRING, and FLOAT; and recovery is handled by Before Image logging.

Note: LFILTER is a layer required for processing cyclic queries. For further details, see the DaTE help menu.



Statistics about the size of the generated DBMS can be obtained by pulling down the **Misc** menu and selecting **Statistics**. A window similar to the one shown at the top of the next page will be displayed:



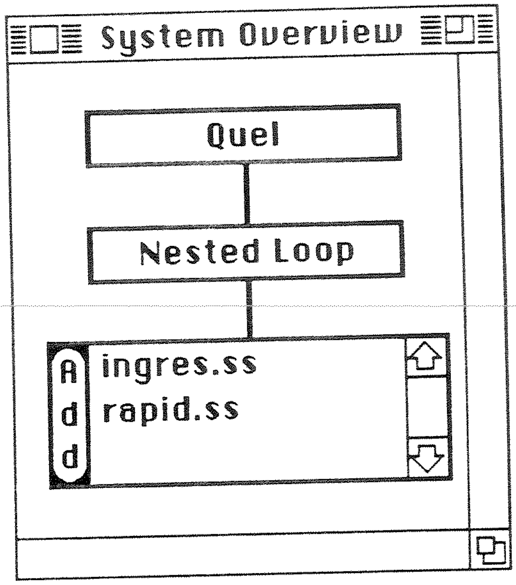
Database Management System Statistics

Manager/Layer	Total	Lines of Code Selected	% of Library
System Managers	6435	6435	100%
System Layers	4863	4863	100%
System Utilities	1427	1427	100%
File Manager	42463	21485	50%
File Layers	6581	1389	21%
Link Layers	7456	982	13%
Model Layers	2468	1234	50%
Totals	71693	37815	52%

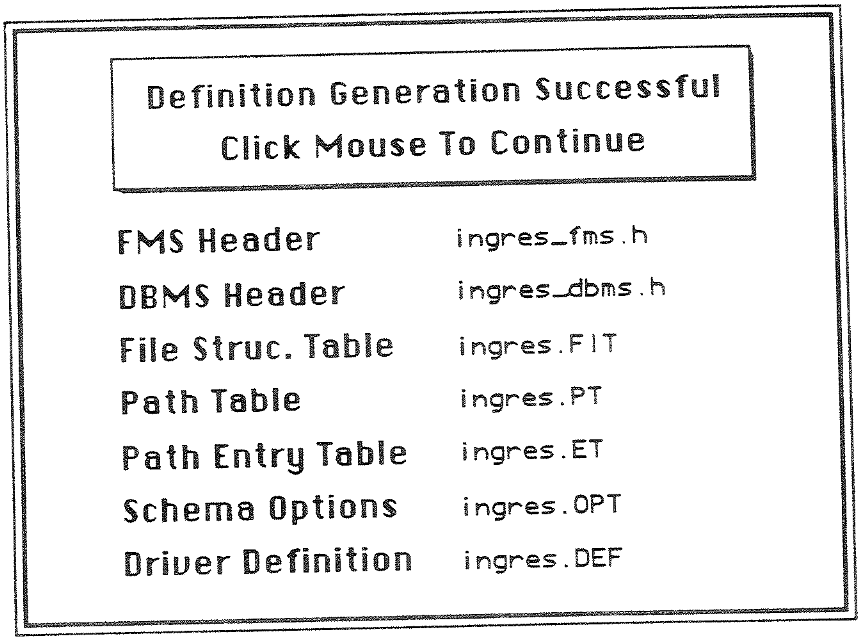
Of the 71K+ lines of code in the Genesis libraries, approximately 52% is referenced in the Ingres DBMS. (Lines that are unreferenced are not included when the Ingres DBMS is assembled). A similar window exists for FMSs.

> Note: **System Managers** and **System Utilities** refer to a standard package of ADTs (queries, into-lists, etc). that are referenced by virtually all DBMS layers. **System Layers** is a generic name given to layers listed on the DBMS's operation bus. **File Manager** refers to FMS code that is generated. **File Layers**, **Link Layers**, and **Model Layers** refer to file mapping, link, and data model layers that are referenced.

A DBMS typically supports only one architecture. However, if one wants the 'union' of several different architectures (to have the capabilities of several individual DBMSs), one can click multiple architectures onto the Architecture Bus. A composite architecture is formed by taking the union of all data models referenced and placing them on a data model bus, the union of all referenced link layers is placed on a link bus, and the union of all storage systems is placed on a storage system bus. The composite architecture of a DBMS can be viewed by pulling down the **Misc** menu and selecting **DBMS Overview** when the DBMS window is active. The union of Ingres.arch and rapid.ss is show in the figure at the top of the next page.



To generate the configuration files of a DBMS, pull down the **File** menu and select **generate**. The following window will be displayed:

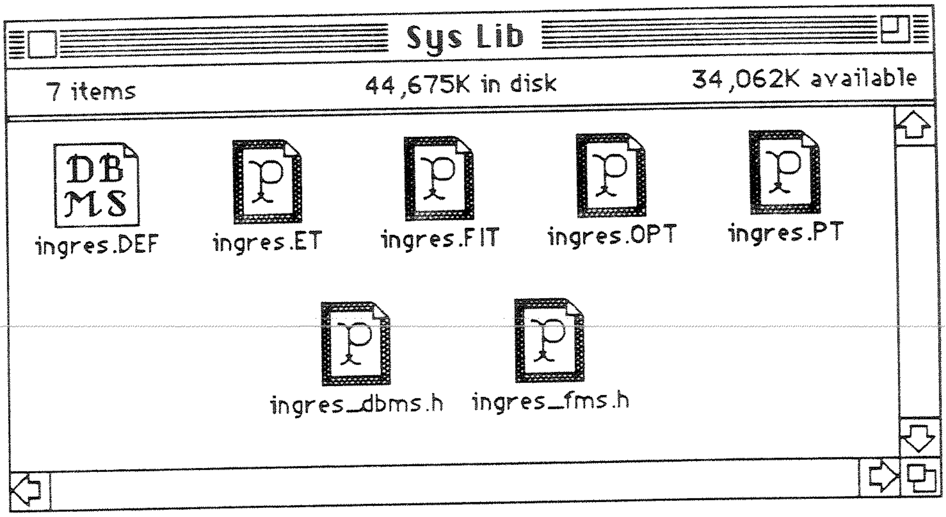


Definition Generation Successful

Click Mouse To Continue

FMS Header	ingres_fms.h
DBMS Header	ingres_dbms.h
File Struc. Table	ingres.FIT
Path Table	ingres.PT
Path Entry Table	ingres.ET
Schema Options	ingres.OPT
Driver Definition	ingres.DEF

The **FMS Header** and **DBMS Header** files are compiled with the Genesis library to produce Ingres. The **File Structure Table**, **Path Table**, **Path Entry Table**, and **Schema Options Table** are read by DBMS executables to process dml operations. The **Driver Definition** is a DaTE readable document that is a copy of the DBMS window that defines ingres. Provided the **Sys Lib** folder is empty, the results of this generation are shown at the top of the next page.



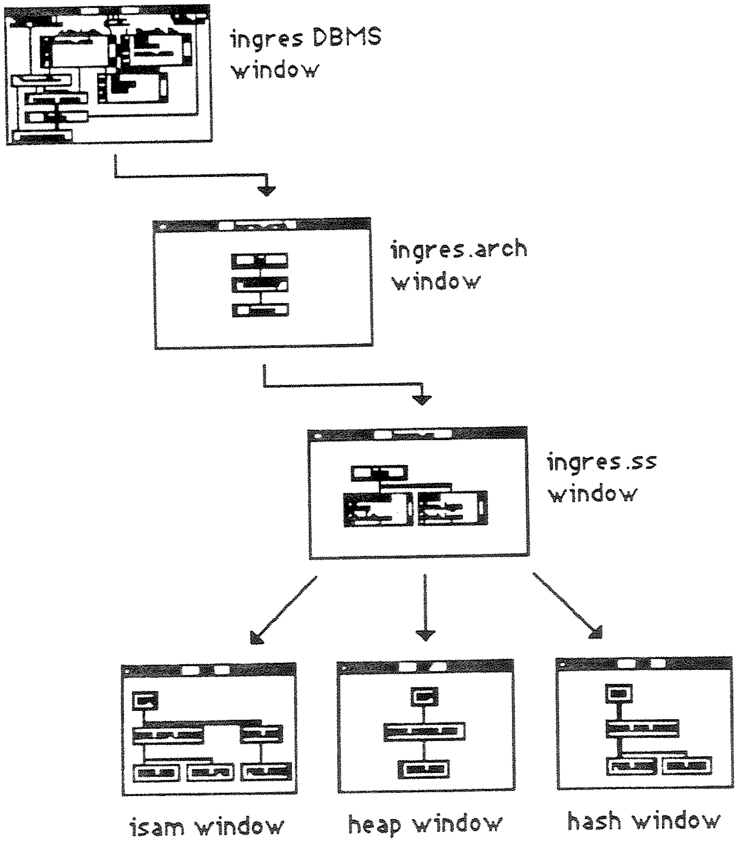
Sys Lib

7 items 44,675K in disk 34,062K available

Note: Again, we recommend that DBMS designs and configuration files be placed in the **Sys Lib** of the DaTE folder.

Window Management and Complex Architectures

Architectures quickly become too large to fit on single windows. This is one reason why DaTE has different windows for different architectures and different systems. DaTE provides a convenient mechanism to navigate among interrelated windows/architectures. The figure below shows relationships between different windows and different parts of a DBMS design:



Recall that complete architectures are treated as unparameterized layers. When one clicks on the box of an architecture and selects **Information**, the window for that architecture is opened. Since architectures can be nested, navigational paths may become long. To retrace to the parent window, pull down the **Windows** menu and select **Back**. Thus, one can navigate the tree of windows of the Ingres DBMS through menu selections.

> Note: At present, we recommend that **at least** one relational architecture be included in a DBMS specification. If a relational architecture is not included, the system that is generated has only a programming language interface. There is no convenient driver that is available, currently, to allow users to explore such systems and issue basic database calls without a considerable amount of programming. If a relational architecture is included, then either SQL or QUEL - a high level query language is made available for issuing basic calls.

Convert Refs

When an architecture is created, there are references to primitive layers and possibly other architectures. DaTE saves these references using the MacIntosh equivalent of absolute pathnames. Thus, if you decide to copy you architectures onto another disk, or simply to move them from one folder to another, you've changed the pathnames of your architectures. DaTE will choke when it tries to read such files again. As a partial fix to the problem, the **Convert Refs** utility is used. Drag it into the folder whose architecture files have been moved. Double-click it to start it to execute. If all pathnames can be converted properly, it will report success. If some pathnames can't be converted, the count of the number of errors is reported. Not exactly useful, but when you run DaTE, it will become obvious which architecture files haven't been properly translated.

Phase 2 - Assembly

A Genesis-produced DBMS has two executables: **Gdefine** and **Gdml**. **Gdefine** compiles schemas and creates files for loading. **Gdml** loads empty databases and supports database processing via nonprocedural query languages. In the following, we will use ingres to illustrate the production of Genesis executables.

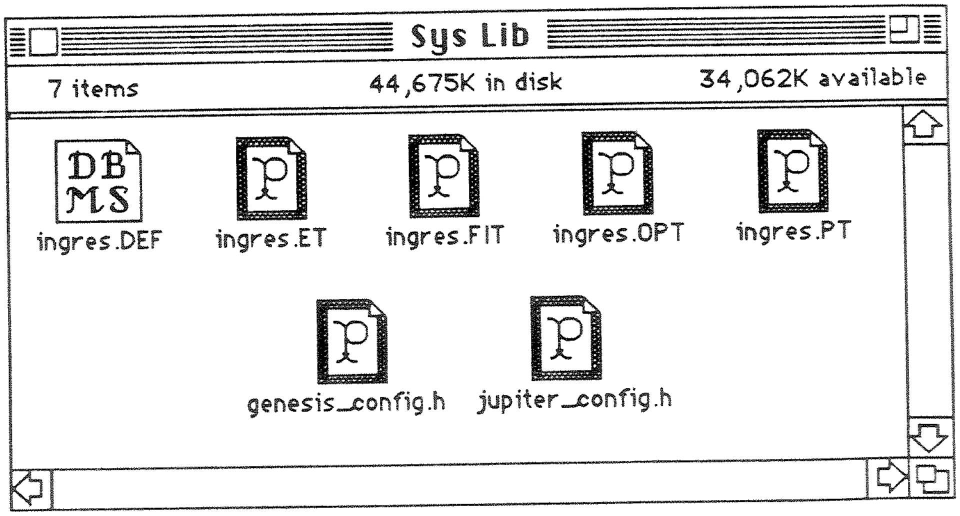
> Note: Ultimately, **Gdefine** and **Gdml** will be merged; it is only for historical reasons that both were developed independently. The compilation of **Gdefine** and **Gdml** takes about 10 minutes. As we'll explore later, Universals can eliminate the need for compilation altogether.

There are two ways to produce executables. Either you build versions of Gdefine or Gdml that containly only the layers that are required for your target system, or you can build a universal Gdefine and Gdml. Universals, as we will call them, are executables that contain virtually every building-block in the Genesis Library. Module interconnections are realized at run-time via dispatch tables. Thus, using Universals, it is possible to go directly from a DaTE specification to DBMS execution, eliminating the need for compilation. Of course, a penalty to be paid for Universals is slightly slower speeds and the inability to implement certain functions. We will explain more about Universals at the end of this chapter.

In the following sections, we'll explain how to compile versions (Universal or otherwise) of Gdefine and Gdml. Producing Gdefine and Gdml executables is a 3-step process.

Step 1 - Renaming Configuration Files

Among the header files that are referenced by Genesis source are **genesis_config.h** and **jupiter_config.h**. Both are transcripts of a DBMS design which partially specify the interconnections between Genesis layers. (genesis_config.h deals with DBMS layers; jupiter_config.h deals with FMS layers). DaTE distinguishes different config.h files by prepending the name of the DBMS. The first step in compilation is to rename ingres_dbms.h to genesis_config.h and ingres_fms.h to jupiter_config.h. Here what the Sys Lib folder should look like after the renaming:



Step 2 - Uploading Configuration Files

The UNIX directory for Genesis has the following contents:

```
GenConfig/    bin/         ddl/            dml/         drivers/
expanders/    headers/     jupiter3.1/     make/        nubs/
obj/          system/      transformers/   utilities/   BigConfig/
```

As a quick overview, the GenConfig/ directory will contain some of the DaTE produced configuration files. The obj/ directory contains the object files of Genesis, bin/ contains Genesis executables, and the rest are directories containing source files. The directory structure of bin/ is:

```
GenConfig/    demo.test*   gdml*        gdefine*
link1.test*   link2.test*  link3.test*  paces/
emp/          remdb*       ut/          valid/
```

For uploading configuration files, the GenConfig/ and the bin/GenConfig/ directories are relevant. The _config.h files should be placed in the GenConfig/ directory. The other files (not including the .DEF file) should be placed in the bin/GenConfig/ directory. The .DEF file is not really needed, and can be discarded.

The reason for having two GenConfig directories is that Genesis executables require certain configuration files to run, while other configuration files are needed only for compilation. The files in GenConfig/ are for compilation. The files needed by Gdefine and Gdml are in bin/GenConfig/. As a rule, if Gdefine and Gdml are moved to a different directory, the bin/GenConfig/ directory must also be in that directory.

Step 3 - Building Gdefine and Gdml

The contents of the make/ directory are:

```
Dml.make      Gdefine.make    README                 Grun.make
Gen.make.h    Target.make.h   Target.make.h.sample
```

As a quick overview, Gdefine.make is the Makefile for Gdefine, and Dml.make is the Makefile for Gdml. The README file explains how to add new software building blocks to Genesis and to include them in the Makefiles. Explanations of the other files are also found in README. To produce Gdefine and Gdml, enter to the make/ directory and type:

```
make -f gdefine.make
make -f gdml.make
```

The Gdefine and Gdml executables will be deposited in bin/.

Universals

It is possible to build versions of Gdefine and Gdml that contain virtually all layers of Genesis. The configuration files for Universals are found in the BigConfig/ directory of the Genesis directory. If you list BigConfig/, you'll see the Universal genesis_config.h and jupiter_config.h files.

Virtually all layers can be included in Universals, but there are exceptions. Universals can only have one recovery layer. So if you want to differentiate Universals that rely on shadowing from those that rely on before-image page logging, you'll have to create multiple versions of the Universals.

Open the jupiter_config.h file and scroll to the bottom. That's where you'll see constants like:

```
/****************       RECOVERY  TYPES       ********************/

#define BFIM_RECOV                1
#define NULL_RECOV                0
#define SHADOW_RECOV              0
#define DBCACHE                   0
```

These commands will define Universals that rely on before-image logging. Altering the use (1) and nonuse (0) values, one can select different recovery implementations. Remember: only one entry should be set to 1; the rest must be zeros.

When you are compiling Universals, discard the genesis_config.h and jupiter_config.h files generated by DaTE and use those in BigConfig. Rename all other configuration files to "demo", as this is the name given in the BigConfig universals. (Actually, to change the name to something other than demo, open up genesis_config.h in BigConfig and edit the #define DB_NAME to whatever name you'd like). Other than this, building Universals is no different than building any other DBMS.

Phase 3 - Usage

The usage phase deals with the customization of DDL directives, creation of schemas, database loading, and database processing. To get started, recall the contents of the bin/ directory:

```
GenConfig/    demo.test*    gdml*         gdefine*
link1.test*   link2.test*   link3.test*   paces/
emp/          remdb*        ut/           valid/
```

As a tour, we created **Gdefine** and **Gdml** in Phase 2; they are the executables of our target DBMS. The directories `paces/` and `valid/`, and the .test files are used to validate **Gdefine** and **Gdml**. (More on this later). The `emp/` and `ut/` directories are two small databases provided with Genesis. The **emp** database consists of a single relation; the **ut** database has three highly interconnected relations.

Phase 3 begins by customizing DDL directives.

Customizing DDL Directives

The .OPT file generated by DaTE lists DBMS-specific directives on how to store different relations and links. DaTE assigns a name for each directive, but the name itself may not be syntactically correct or easily rememberable. Customizing DDL directives is, in effect, providing alternate names.

The names DaTE generates reference labels given to layer parameters or user-defined architectures. Any sequence of characters can serve as a name for DaTE. However, the current Genesis DDL only allows names to begin with a letter, followed by a sequence of letters, digits, or underscores (_). Illegal names must be repaired by editing.

As an example, open **ingres.OPT**, which is in the `bin/GenConfig/` directory. You'll see the following text:

```
Row        Tag   Next   Type   Path   Fit   Name
#
  0          1    -1     1      0     -1    ingres.ss
  1          2     2     0     -1      1    hash_index
  2          4     3     0     -1      2    heap_index
  3          8    -1     0     -1      0    isam_index
  4         10     5     0     -1      0    isam_internal
  5         20     6     0     -1      1    hash_internal
  6         40    -1     0     -1      2    heap_internal
```

The only column of interest to us is `Name`. (Don't change any other entries!!). The label `ingres.ss` should be familiar; it is the name of the ingres storage system that we defined in Phase 1. We need to rename this label, to say '`ingres`', because it has an illegal character (i.e., the dot):

```
  0          1    -1     1      0     -1    ingres
```

All other names are acceptable as is.

Once the Name column contains legal names, further renaming can be motivated by examining the semantics of each label. Schema-defined relations that are to be stored via the ingres storage structure must be tagged with the option 'ingres'. When there is only one storage system - as in our case - using explicit storage system tags is unnecessary. Tags are needed if relations could be mapped by several storage systems. (Recall that multiple storage systems are possible when a DBMS has supports multiple architectures). We'll illustrate tagging shortly.

The next three options - hash_index, heap_index, and isam_index - are directives to store index files in hash, heap, or isam structures. The last three options - isam_internal, hash_internal, and heap_internal - are directives to store data files in isam, hash, or heap structures. Again, we'll explain how they are used in schemas shortly.

We'll shorten the labels in rows 1-6 by dropping _internal suffixes and replacing _index suffixes with 'x', yielding:

Row #	Tag	Next	Type	Path	Fit	Name
0	1	-1	1	0	-1	ingres
1	2	2	0	-1	1	hashx
2	4	3	0	-1	2	heapx
3	8	-1	0	-1	0	isamx
4	10	5	0	-1	0	isam
5	20	6	0	-1	1	hash
6	40	-1	0	-1	2	heap

In general, any name for a label can be used as long as it is unique within the table and does not conflict with Genesis DDL reserved words. These words are listed below:

int
cstring
vstring
float
double
byte
files
links
database
rpg
set
char
index
primary_key
ring_list
ptr_array

Defining Schemas

Open the file **ut.schema** in the `ut/` directory. It is shown below with labels in bold:

```
DATABASE ut {

FILES

      employees  {
         empno          INT                     primary_key;
         age            INT                    ;
         dept_name      CSTRING ( 20 )         ;
         name           CSTRING ( 22 )            primary_key indexed isamx;
      }  heap ingres;

      dept {
         deptno         INT                     primary_key;
         dept_name      CSTRING ( 20 )          indexed hashx;
         chairman       CSTRING ( 22 )         ;
      } isam;

      prof {
         profno        INT                      primary_key;
         prof_name    CSTRING ( 22 )           ;
         department   INT                      ;
      }  hash;


LINKS
      /* 1:n links */
      P_worksin    : dept.deptno = prof.department             /* ring_list */;
      E_worksin    : dept.dept_name = employees.dept_name      /* ring_list */;

      /* 1:1 or 1:0 link */
      P_empdata    : prof.prof_name = employees.name           /* ring_list */;
      P_chairdata  : prof.prof_name = dept.chairman            /* ring_list */;
      E_chairdata  : employees.name = dept.chairman            /* ring_list */;
}.
```

Three relations are defined plus five links. The syntax for relations is straightforward. Link syntax is <name> <colon> <join-predicate> <options>. Join predicates must be equality-based with no disjunctions. Links interrelate a pair of different relations. One is the **parent** and the other is the **child**. Parentage is conveyed by phrasing of the join predicate; the first relation referenced is the parent. Thus, in P_worksin, the dept relation is the parent and prof relation is the child.

DDL directives are options that can adorn relations, individual fields, and links. In addition to those in ingres.OPT, there are a few options that are reserved: **indexed**, **ring_list**, **primary_key**, and **ptr_array**. **indexed** is used to tag fields that are to indexed; **primary_key** tags fields that define the primary key of a relation; **ring_list** and **ptr_array** tag links to specify that their implementations are to be ring lists or pointer arrays. Note that only **primary_key** is provided to all Genesis-produced DBMSs. **index**, **ring_list**, and **ptr_array** are available only if their corresponding layers are present in the DBMS.

To see how these directives are used, look at the employees relation:

```
employees   {
   empno          INT                        primary_key;
   age            INT                        ;
   dept_name      CSTRING ( 20 )             ;
    name            CSTRING ( 22 )             primary_key indexed isamx;
  }  heap  ingres;
```

The field pair (empno, name) is declared to be the primary_key of employees. The name field is to be indexed, and its index file is to be stored in an isam structure. The employees relation is to be stored in a heap. The option 'ingres' adorns employees to tell **Gdefine** that employees is to be stored via the ingres storage system. (As mentioned earlier, the use of the 'ingres' tag in this example is not necessary). Normally, when one specifies a field to be indexed, one also must specify how the correponding index file is to be stored. If multiple options are available, but no option is given, **Gdefine** will make a choice and will report its choice during schema compilation.

Now look at the P_worksin link:

```
P_worksin    : dept.deptno = prof.department          /* ring_list */;
```

If the ingres DBMS supported ring lists, the **ring_list** tag would be placed as shown, but not within comment markers. The P_worksin declaration above is untagged. An untagged link tells **Gdefine** that the link must be implemented by a join algorithm.

Note: If there are no link layers, the current version of **Gdefine** will still recognize links. It is only during Gdml execution that references to the link will be recognized as an error.

Defining Databases

All information about a database is stored in a directory of the same name. It must be in the same directory as **Gdefine** and **Gdml**. As an example, list the ut/ directory:

```
UT.SCHEMA    ut.dept.data  ut.prof.data ut.employees.data
```

The schema is present along with three other files. These files, respectively, contain the raw data that is to be used to load the dept, prof, and employees relations. Their format is straightforward. Each line contains data for a record. Column values are separated by commas with no embedded blanks; strings are enclosed within quotes. The naming of raw data files is important. If s is the name of a schema and r is the name of a relation in a schema, then the name of the raw data file is 's.r.data'. Thus, ut.dept.data is the raw data file for the dept relation in the ut database.

When **Gdefine** is run, the following output is generated:

```
***** Creating Volumes *********

:ut:ut created

***** Creating Files ***********

employees created
$emp!name created
dept created
$dep!dept_name created
prof created
```

The names of each volume, conceptual file, and internal file that is created is listed. Internally generated files have names prefaced by $. Index file names, for example, take the first three characters of the conceptual file, followed by bang (!), followed by the name of the field. Thus $emp!name is the name given to the name index file of the employees relation.

Gdefine produces a large number of additional files that will appear in a database directory. These files are used by **Gdml** to know how to perform conceptual-to-internal mappings of relations and links in the database. When **Gdml** is run for the first time on a newly compiled schema, it will look for a raw data file for each relation, and will load that relation. Hell breaks loose if the file isn't found. Subsequent runs of **Gdml** will not invoke database loading.

Note: To dump a database, one simply has to execute a "select * from relation" and route the output to a text file. The format used in **Gdml** to output tuples is the same format it uses for raw data file loading.

Validation

A set of SQL scripts has been developed to validate Genesis produced DBMSs against the emp and ut databases. For the emp database, the script is **sfile.test**. For the ut database, three scripts are available: **link1.test** tests multifile retrievals, **link2.test** tests multifile updates, and **link3.test** tests cyclic queries. These files are in the bin/ directory along with Gdefine and Gdml.

As an example, suppose the UT database has been loaded (and has not been modified). To run link1.test, just type:

```
link1.test
```

If nothing is output, then the SQL tests in link1.test all succeeded. This should always be the case. If something is printed, make sure that the UT database has not been modified prior to running the test. If link1.test still fails, there is a bug in Genesis. Please report such bugs immediately along with documentation and scripts to reproduce the bug.

All other tests are conducted in the same way.

Installation

Genesis comes in two parts: DaTE, the Genesis DBMS Layout Editor, and the Genesis Source. The Source comes on a 45MB Sun-readable tape cartridge, while DaTE comes on a 1.4MB disk. Mac IIc series can read this disk; Mac II's cannot. So if you are getting 'unreadable disk' errors, then most likely your drive can't read 1.4MB disks.

The installation of DaTE is accomplished just by copying the disk onto a hard disk. Genesis Source will require a bit more work. You'll need to create a special directory, we'll call it GENESIS/ here for discussions, and type:

```
cd GENESIS/
tar xvf <tape-drive>
```

to extract the contents of the tape.

Once GENESIS/ is in place, you'll have to go and change three (3) make files in GENESIS/make. Let's look at the contents of this directory:

```
Gdefine.make            Gdml.make    Gen.make
Grun.make               README       Target.make.h
Target.make.h.sample
```

Gdefine.make creates Gdefine and Gdml.make creates Gdml. Target.make.h is a file that is #included in both Gdefine.make and Gdml.make. It contains the absolute pathname of directory in which Gdefine and Gdml is to be placed.

You must modify Gdefine.make and Gdml.make by changing ROOT_DIR from /v/yog/v1/genesis/G2 to GENESIS/:

```
ROOT_DIR = GENESIS/
```

Again, this must be done in both Gdefine.make and Gdml.make. The other modification is to set TARGET_DIR in Target.make.h from /v/yog/v1/genesis/G2 to GENESIS/:

```
TARGET_DIR = GENESIS/
```

This will place the Gdefine and Gdml executables in GENESIS/bin.

What are all the other files? Should you wish to experiment with existing layers by changing them, or by adding your own layers, you do this by modifying Target.make.h according to the instructions in file README. An example is given in Target.make.h.sample.

Gen.make.h is one of three make-include files that define how to compile Genesis source. The other include files are GENESIS/system/make/Sys.make.h and GENESIS/jupiter3.1/make/Jup.make.h. The Sys.make.h file deals with generic system utilities, such as available list routines (which handle dynamic memory allocation), avl_tree routines (used in grid file retrievals), etc. Jup.make.h deals with file management related source, like shadow recovery algorithms (shadow.h), file structure algorithms (filemgr1.c, filemgr2.c), and so on. Gen.make.h deals with the layers of Genesis that lie above file management, like secondary indexing (seg_index.c, and so on). Unless you are modifying Genesis source, you need not ever have to change these make files.

Finally, Grun.make is a procedural interface to Genesis that, quite frankly, we no longer use. It was used for debugging and running Genesis operations (cursor creation, initialization, etc.) one at a time. However, we've found a more effective way to debug Genesis was through an SQL or QUEL interface. So Grun.make may disappear altogether in subsequent

Database Processing

A Genesis-produced DBMS can run subsets of SQL, QUEL, or both. The Gdml prompt 'SQL:' requests a SQL command; 'QUEL:' prompts for a QUEL command. Typing SQL will switch the DBMS to accept SQL commands; typing QUEL switches to QUEL commands. (Switching is possible only if a DBMS supports both interfaces).

A subset of SQL and QUEL is supported. This doesn't include, unfortunately, nested selects and aggregations. So too are join predicates within update and delete statements. (That is, update and deletion predicates are restricted to reference a single relation).

Note: these restrictions were imposed simply to keep this version of Genesis tractable. There is no apriori reason that prevents full-blown SQL and QUEL to be supported.

Link names can appear in SQL predicates in place of their schema-defined join predicate. Thus, the following queries are identical:

```
select *                                    select *
from   dept, prof                           from   dept, prof
where  dept.deptno = prof.department        where  P_worksin
```

If aliases are used in an SQL select, link names cannot be used. (The reason is that link names are bound to schema-defined names of relations, not aliased names). This means that link names cannot be used in QUEL retrieves, because of the standard use of aliasing relations via RANGE-OF commands.

If a dml statement becomes too long to fit on a single line, lines can be continued by a backslash (\) carriage return. Although not shown, the first two lines of both of the above select statements were terminated by a \.

QUEL has been extended to support ORDER_BY clauses to order the output of retrieval statements. The syntax is the same as that for SQL. Another simple extension is the * feature for retrieval:

```
retrieve (E.*) where...
```

where 'E.*' is a shorthand for all fields of relation E.

To exit from Gdml, type 'exit' or 'quit'.

releases.

To summarize, changing three lines in three different files is all that should be needed to get Genesis up and running.

Features and Known Bugs

DaTE. Not everything is perfect. DaTE has bugs that trash memory or make illegal memory references. Unfortunately, some of these bugs are very difficult to track down and fix, due to the memory management scheme of the MacIntosh. Simply put, the errors aren't always reproducable. You'll discover the do's and don'ts on your own; it isn't difficult to make DaTE work for you. At least with a bit of alchemy. I highly recommend that you install a version of **MacsBug** to trap these errors.

DaTE tries to check for design errors. Most of the time, it catches mistakes and reports them to you. However, if you create a storage system which references itself, don't expect things to work.

We've already alerted you to DaTE's other quirk - the need for the Convert Refs application. See the end of Phase 1 for an explanation.

The size (in terms of number of lines) of each layer, as registered in DaTE is not accurate. We are in the process of determining how to count line numbers (should comments be included, etc.) and will be updating these numbers shortly. Yes, there are 70,000+ lines of Genesis source, but removing headers and comments (which are essential to understand what's going on) will reduce this number.

Finally, don't mess with trying to define new layers or altering existing layer definitions, unless you know what you are doing. Technical reports exist which explain the basic ideas. Once you know what each button, etc. means, then you can proceed to experiment.

Genesis Source. You may have noticed that there is source code or references to a db_cache recovery, grid file structure, and references to Ziv-Lempel encoding. Right now, db_cache and grid have some bugs which we are trying to track down. Ziv-Lempel encoding does work on Sun3s, but does not work on Mac IIs. A port revealed that the Ziv-Lempel algorithms we are using require enormous byte arrays, larger than Think C can handle. So we eliminated it.

Also, B+ trees are known not to work when at most two records fit per node/block. Debugging B+ tree code is painful, and this is still on our to do list.

Gdefine & Gdml. Again, nothing is perfect. Gdml somehow leaks memory. A consequence of this is that very long scripts will not execute to completion before Gdml blows up. You can go quite some time before something happens, but something will happen eventually.

Another problem is that presently, all genesis databases are stuffed into individual volumes of a single, predefined size. For Mac IIs, the size is 2000 blocks of 512 bytes. To change this, you'll need to edit genesis_tun.h, which is subfolder **headers** in the **GenSource** folder. As a general rule, and header file (genesis, jupiter, or system) _tun.h contain constants that you can change. Just be careful.