MODELING THE PHYSICAL STRUCTURES OF COMMERCIAL DATABASE SYSTEMS

D. S. Batory

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

TR-83-21 September 1983

Abstract

Modeling the physical structures of a DBMS is a prerequisite to understanding and optimizing database performance. Previously such modeling has been difficult because some fundamental principles of physical database design and implementation have not been well understood.

Functional equivalence - the idea that different structures implement the same concept - is introduced in this paper. It enables the physical structures (and operations performed on them) to be modeled at different levels of abstraction. The basic techniques of the approach are elementary transformations which are rules for mapping from one level of abstraction to lower, more concrete levels. Functional equivalence enables the complex storage structures of commercial databases to be modeled in a simple, precise, systematic, and comprehendible way. Models of the INQUIRE, ADABAS, INGRES, and SYSTEM 2000 physical architectures are presented.

1. Introduction

Improving the performance of commercial database systems is a significant and very difficult problem. Progress toward its solution has come from the development of models of physical databases. Such models have been used to study the performance and optimization of hash-based files, indexed-sequential files, B+ trees, inverted files, transposed files, hierarchical and network databases, and query processing, among others.

Since 1970 there have been significant advances in modeling physical databases. These advances, as a general rule, have been encorporated into the so-called general models of physical databases. Hsiao and Harary ([Hsi70]) were among the first to propose a general model which described a spectrum of file structures by means of a small collection of parameters. Severance ([Sev72]) extended this approach by introducing the concept of pointer vs. sequential linkages in modeling file implementations. Senko, Altman, Astrahan, and Fehder ([Sen73]) advanced similar but more general ideas in their data independent accessing model (DIAM). Strings and basic encoding units were seen as primitive components of database implementation. Yao ([Yao77]) extended Severance's work by identifying file structures with directed trees. More recently, March, Severance, and Wilens ([Mar81]) demonstrated the frame to be a basic unit of physical database construction, and Batory and Gotlieb ([Bat82]) showed that complex physical database structures and operations could be modeled simply by decomposition.

In spite of recent advances, there still are no models of physical databases that can account for the diversity and complexity of structures (and their associated algorithms) found in commercial DBMSs in a comprehendible way. Although existing models have been used as starting points, considerable

effort is needed to adapt and extend them just to describe the structures and operations of a single DBMS ([Cas81]). In view of these shortcomings, it is easy to understand why there are so few design and performance aids for commercial database systems.

The difficulties in using current models clearly suggests that some fundamental principles of physical database design and implementation are not well understood and are inadequately represented. A closer look at these models reveals that they generally exhibit two distinct components: a descriptive component, which models the physical structures and operations that underly a DBMS, and a quantitative component, which predicts the performance of the DBMS under projected workloads. The difficulties that we have noted in applying current models can be traced to inadequate descriptive modeling techniques. Unfortunately, to improve the descriptive power of existing models does not simply involve enlarging the spectrum of structures and operations that they describe. It requires much more.

The thrust of our research is to demonstrate some new descriptive modeling techniques which we feel will help bridge the gap between physical database theory and practice. The scope of our work is intended to be quite broad. The goal is to develop techniques which are capable of accurately modeling the underlying structures and operations of a wide spectrum of commercial and specialized DBMSs. As evidence of the generality of the techniques developed sofar, we will show how the diverse and complex physical structures that underly four DBMSs - namely, INQUIRE, ADABAS, SYSTEM 2000, and INGRES - can be modeled in a systematic, precise, and simple way. Our approach is based on a new modeling concept called functional equivalence. It allows the physical structures and operations of a DBMS to be modeled at different levels of abstraction.

We believe this research provides an essential step toward the development of practical design and tuning aids for commercial DBMSs. We also believe that it reveals a way to extend and simplify current methodologies for DBMS implementation ([Tsi77b],[Bar81]) and shows how disparate implementation "tricks" can be related in a formal and comprehensive setting. These and other contributions will be discussed in Section 5.

In the following section, we present an example which underscores the disparity between current theory and practice. It will also help to motivate and explain the concept of functional equivalence.

1.1 The Concept of Functional Equivalence

Consider how index records of inverted files are described in theory and how they are realized in practice. Database texts and research papers define the contents of an index record as a data value and an inverted list containing a variable number of pointers ([And76], [Kro77], [Bat82], [Dat82]). The implicit structure of an index record is shown in Figure 1.

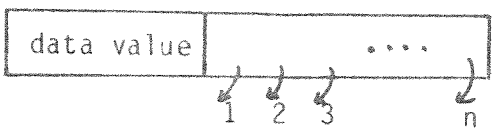


Figure 1. An Abstract Index Record

Commercial and specialized database systems rarely implement index records directly as in Figure 1. The reason is that the underlying file

structures of most databases require records to have a fixed and uniform length; index records of Figure 1 have variable lengths. What Figure 1 really represents is an abstraction of an index record. That is, at some level of abstraction all index records of inverted files have the format of Figure 1. Henceforth, we will refer to such records as abstract index records. Actual database systems materialize abstract index records in a variety of different ways. Here are some examples.

IMS ([IBM81]) and RAPID ([Sta81]) materialize an abstract index record by pairing the data value with each pointer in the inverted list. Each (data value, pointer) pair defines a "concrete" index record, i.e., a record that is actually stored (Fig. 2).

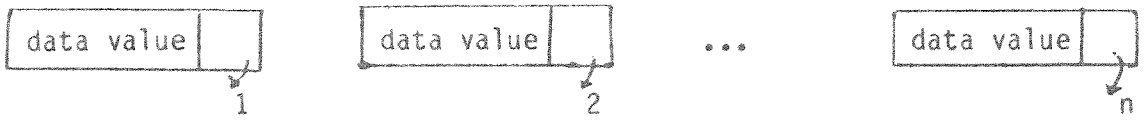


Figure 2. IMS and RAPID Realization of an Abstract Index Record

System 2000 ([Kro77], [Cas81]) materializes an abstract index record by storing the data value in a separate record and the inverted list in one or more additional records. A linear list chains these records together (Fig. 3).

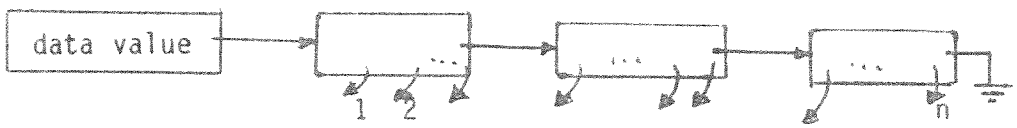


Figure 3. System 2000 Realization of an Abstract Index Record

MRS ([Kor79]) materializes an abstract index record similar to System 2000, except that the first pointer of the inverted list is stored in the record containing the data value (Fig. 4). This was done so that if the data value was an identifier (i.e., a primary key), no inverted list records would need to be accessed.

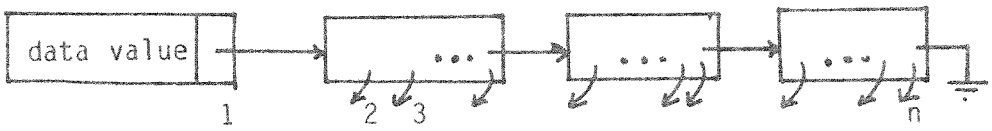


Figure 4. MRS Realization of an Abstract Index Record

It would be quite difficult to describe each of the above materializations accurately using existing models of physical databases without actually altering the models themselves. Moreover, current models would not reveal that each of the above materializations are functionally equivalent. That is, they all realize the same concept: an abstract index record.

Functional equivalence - the idea that different structures implement the same concept [1] - has a much broader application than this simple example suggests. Specifically, the general modeling approach is to start with the generic logical record type that is supported by a DBMS. This generic logical type is an abstract record whose materialization is to be determined. The materialization is specified as a multistep derivation where each step involves one or more elementary transformations. An elementary transformation

[1] Keep in mind that performance considerations play no role in functional equivalence. As can be seen in Figures 2-4, several structures can be functionally equivalent and yet have very different performance characteristics.

maps abstract records to "less" abstract records or concrete records. The result of applying a well-defined sequence of transformations to the generic logical record type is the set of concrete record types and their interconnections that underly the structures of the DBMS. As will become clear shortly, our approach models physical databases at multiple levels of abstraction. It also provides a firm solution to the problem of logical to physical (i.e., conceptual to internal) mappings.

The transformation techniques that we will introduce in this paper are unlike any techniques presently used to model physical databases.[2] Not only will we demonstrate that they provide a necessary means to model actual DBMSs easily, they will also help to make explicit certain fundamental principles of physical database design and implementation which previously were not well understood. A formal model is presented in the following section.

The starting point of our research is the unifying model (UM) of Batory and Gotlieb ([Bat82]). The UM is the only model of physical databases that is expressly based on decomposition, a concept that is essential to modeling operational DBMSs. Physical databases can be decomposed into a collection of simple files and linksets. A simple file is a structure that organizes records of one or more files. Classical simple files include hash-based, indexed-sequential, B+ trees, and unordered structures. A linkset is a structure that relates records of one or more files. Classical linksets include pointer arrays, inverted lists, ring lists, and cellular multilists. By decomposing a physical database and specifying the implementation of each of its constituent simple files and linksets, the implementation of the physical database is defined. These ideas and terms are used extensively in the following sections.

---

[2] The proposed techniques are similar, but not identical, to transformation techniques used in database and program conversion (see [Fry74], [Hou77], [Nav76], [Su81], [Shn82]).

2. Elementary Transformations

The basic modeling techniques of functional equivalence are elementary transformations. They are essentially rules for mapping structures from one (higher) level of abstraction to another (lower, more concrete) level. The elementary transformations presented here were discovered as a natural consequence of modeling the physical architectures of SPIRES ([Sta73]), TOTAL ([Cin79]), MRS ([Kor79]), IDMS ([Cul81]), IMS ([IBM81]), ADABAS ([Ges76]), INQUIRE ([Inf79a]), RAPID ([Tur79]), ALDS ([Bur81]), CREATABASE ([NDX81]), and SYSTEM 2000 ([Cas81]). Ten different elementary transformations are known. Nine are defined and illustrated in this paper; a tenth will be mentioned. Although there is ample evidence that these are the most common transformations, no claim can yet be made about the completeness of this set. We shall address the completeness issue later in Section 4.

To define and illustrate the effects of elementary transformations and to model physical database architectures graphically, diagrammatic notations will be needed. As the nature of our work deals with logical to physical mappings, the problem arises as to which logical data model (E-R, hierarchical, semantic network, relational, etc.) should be used, since each has their own diagrammatic notations. It turns out that the choice of logical data model is not critical to our work. We will be dealing with entities (record types), attributes (fields), and relationships (linksets), concepts that are common to all major logical models ([Nav76]). We will use simple data structure, field definition, and illustration diagrams. If necessary, all of these diagrams can be adjusted to fit the notation of a particular logical model.

Data structure diagrams (dsd) are used to show record types and their relationships. Record types will be denoted by boxes and linksets by

arrows. Abstract record types and linksets will be indicated by dashed outlines; concrete record types and linksets will have solid ones. Figure 5.dsd shows that an abstract record type W is materialized by the record types F and G and linkset L. The direction of the arrow shows F is the parent of G.

Pointers to abstract records naturally arise in most derivations. These pointers, however, must ultimately reference concrete records. To indicate how pointer references are transformed, we rely on the orientation of record types within a dsd. The orientation of F and G in Figure 5, for example, shows that F dominates G. This will mean that a pointer to an abstract record of type W will actually reference its corresponding concrete record of type F. (For almost all transformations, there is a 1:1 correspondence between abstract records and their dominant concrete records; the only known exception (to be discussed later) is full transposition). The dominant concept is recursive; that is, if F records are abstract, a pointer to an F record will reference its dominant concrete record, and so on. In this way, pointers to abstract records are mapped to concrete records.

Field definition diagrams (fdd) are used to define the fields of records. Figure 5.fdd shows that record type F consists of fields $F_1 \ldots F_n$ and P_L; G consists of fields $G_1 \ldots G_m$ and C_L.

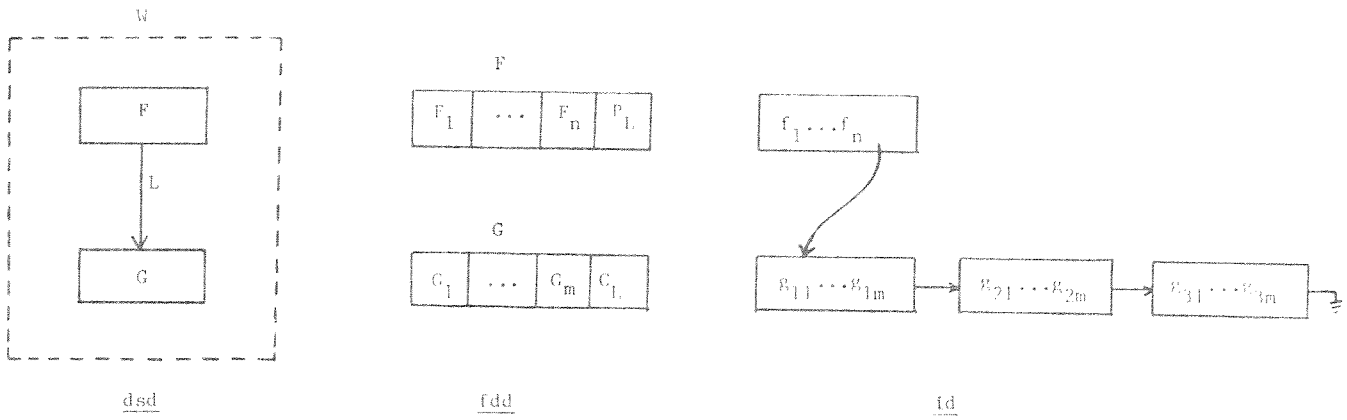


Figure 5. Diagrammatic Notations

As a general rule, whenever a linkset relates two or more record types, some physical structure that implements the linkset is present in the parent record type, the child record types, or both. Such structures (i.e., pointers, counters, etc.) are stored in fields that are specifically allocated for this purpose. In Figure 5.fdd, record F has field P_L (called the parent field of linkset L) which contains the parent record structures of linkset L. Child record structures are stored in field C_L (called the child field of linkset L) in record G. Depending on L's implementation, P_L or C_L need not be present. Both fields are present if L is a multilist. C_L is absent if L is an inverted list. P_L is absent if L is implemented only by parent pointers. (RAPID, ALDS, and CREATABASE use such linksets.)

Illustrations of record and linkset occurrences will be shown in instance diagrams (id). Figure 5.id shows an occurrence of linkset L which consists of a single F record and three G records. Although L could be implemented by one of any number of linkset implementations (e.g., pointer array, ring list, etc.), in this particular Figure L is a multilist. Note that the pointer from the F record to the first G record is stored in field P_L (of Fig. 5.fdd) and the pointer that links one G record to the next is stored in field C_L. So that instance diagrams are readable, 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.

With the aid of these notations and the generic linkset structures of the UM serving as a basis, nine of the ten known elementary transformations are defined in the following paragraphs. The tenth transformation - horizontal partitioning - will be dealt with in Section 5. Examples are given to indicate the utility of each transformation. It should be kept in mind, however, that these examples are not their only possible uses.

Augmentation (of metadata). Metadata can be added to an abstract record. It can be a delete byte, a delete bit, or a record type identifier. It may be stored in a separate field or added to an existing field. In any case, the metadata must be well-defined and have an obvious value. A metadata field is given a name so that it may be referenced later.

Figure 6 shows the data structure diagram and field definitions of an abstract record. Figure 7 shows the result of augmenting metadata field M to this record. RAPID, INQUIRE, ADABAS, and SYSTEM 2000 use augmentation.

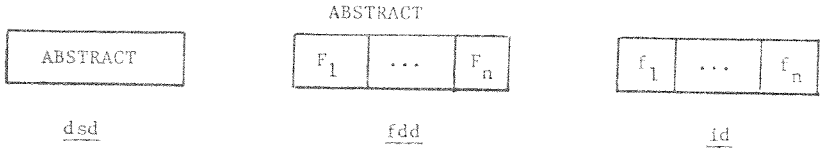


Figure 6. An ABSTRACT Record Type.

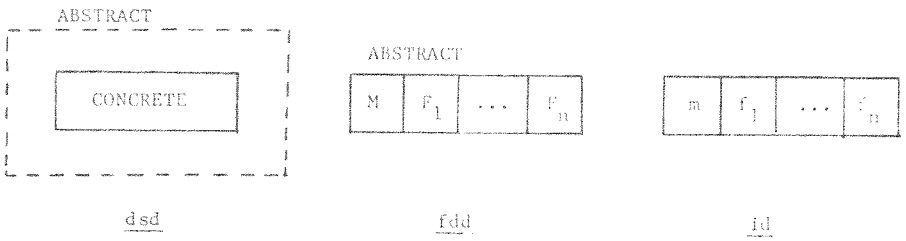


Figure 7. Augmentation of Metadata.

Encoding. Abstract records or selected fields thereof can be encoded. The motivation for encoding might be for data compression - to reduce file storage requirements - or data encryption - to protect confidential information. 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 ([Wel72])), and Huffman encoding ([Huf52]). Well-known encryption algorithms are block ciphers ([Sha77]) and the NBS data encryption standard ([Nat77]).

Figure 8 suggests the result of encoding the abstract record of Figure 6. ADABAS, IDMS, INGRES, and SHRINK/2 ([Inf78]) use encoding. All four employ compression algorithms; SHRINK/2 additionally supports data encryption.

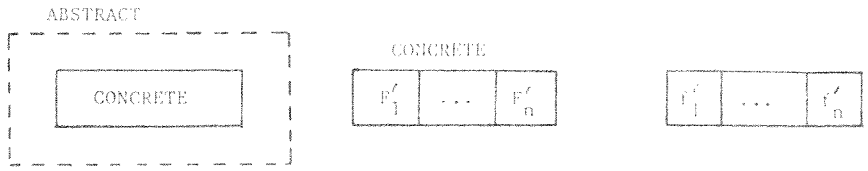


Figure 8. Encoding.

Extraction. Indexing a field of an abstract record is an example of extraction. The general idea is to extract the set of all distinct data values that appear in specified fields of abstract records of one or more given types.[3] Normally, one field per record type is extracted for each application of the transformation. An index record type is created in the process, where each extracted data value is stored in a distinct index record. Each index record is related to all abstract records that possess its data value. The index record type is connected to each abstract record type by a separate linkset. Figure 9 shows the result of extracting field F_i from the abstract record of Figure 6. As a general rule, index record types are not dominant.

There are two known variations of extraction. Figure 9 illustrates extraction with duplication where the extracted field F_i appears in both the $INDEX_i$ and CONCRETE record types. ADABAS, MRS, SYSTEM 2000, and INQUIRE use extraction with duplication to create indices on data fields.

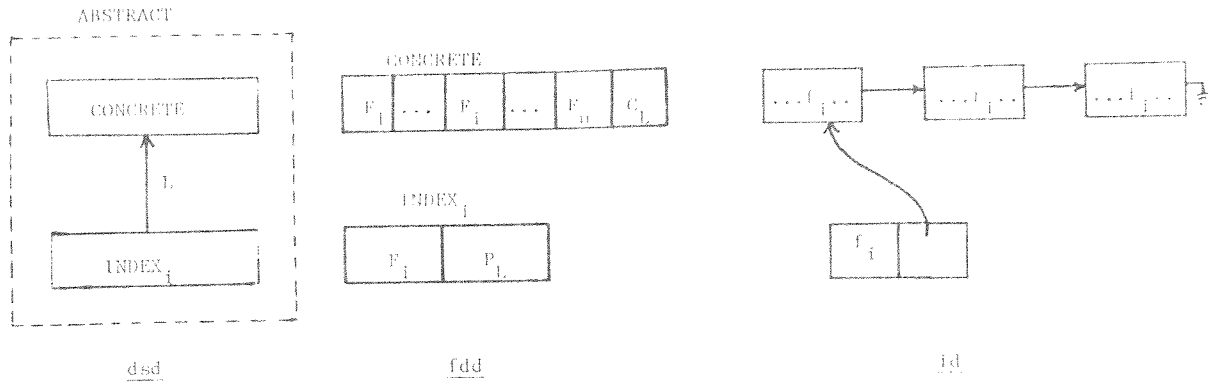


Figure 9. Extraction with Duplication

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

The other variation is extraction without duplication, i.e., the extracted field does not appear in the CONCRETE record type. Figure 10 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 define the domain of F_i; there are no pointers or linkages which connect a data value of the dictionary to all of its occurrences in abstract records. (In contrast, an index has such linkages). Note that linkset L in Fig. 10.id is implemented only by parent pointers. CREATABASE, ALDS, and RAPID use extraction without duplication to create dictionaries on data fields.

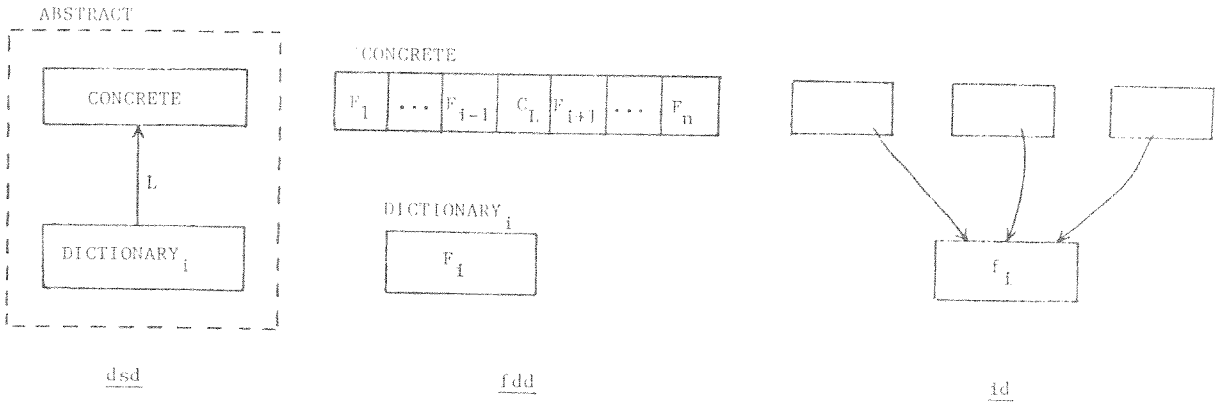


Figure 10. Common Dictionary Configuration

Collect. The DBTG concept of a singular set, which links together all records of a given type, is an example of the collect transformation. The general idea is to collect all instances of one or more abstract record types together onto a single linkset occurrence. Figure 11 shows the result of applying the collect transform to the abstract record of Figure 6. Linkset L is a multilist (with precisely one occurrence) in Figure 11.id. Note that the parent record structure of L is maintained as part of the metadata of a DBMS. (This metadata is indicated by "*" in Figure 11). INQUIRE, DMS-1100, and SYSTEM 2000 use this transformation.

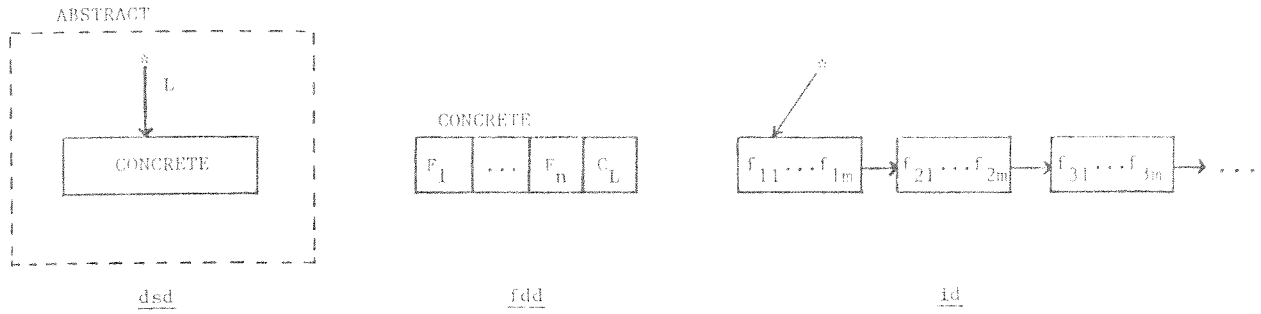


Figure 11. Collection.

Segmentation. Abstract records can be partitioned along one or more field boundaries to produce two or more subrecords. One subrecord is designated as the primary (or dominant) record, the rest are secondary records. A linkset connects the primary record type to each secondary record type.

Segmentation can occur with or without duplication of data fields. Figure 12 shows the segmentation of fields $F_1 \ldots F_k$ from $F_{k+1} \ldots F_n$ of the abstract record of Figure 6. No fields are duplicated and linkset L is a singular pointer (i.e., a pointer array with precisely one pointer) with parent pointers. The same segmentation occurs in Figure 13, except that field F_j is duplicated. RAPID and IMS use segmentation with duplication; ALDS and ADABAS use segmentation without duplication.

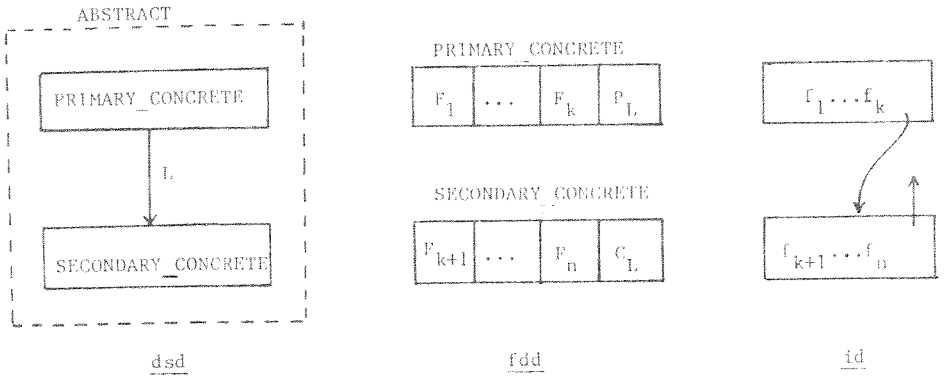


Figure 12. Segmentation without Duplication.

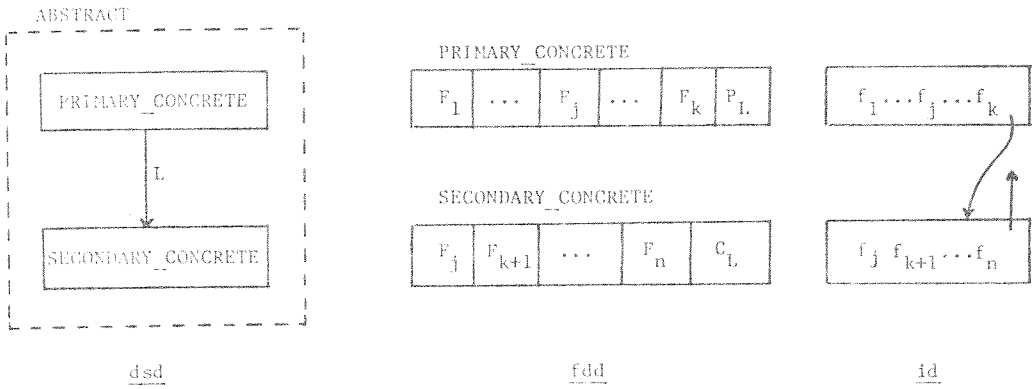


Figure 13. Segmentation with Duplication.

Two instances of segmentation without duplication are so well known or occur so frequently that they are given special names. Full transposition segments each field from all other fields. That is, if there are n fields in an abstract record type then a full transposition produces n record types, each containing precisely one field (see Fig. 14). Because all fields are treated identically, the resulting record 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 linkset L of Figure 14.dsd, which interconnects the n record types, is drawn in a way which does not require the designation of 'parent' and 'child'. Further information on transposed files can be found in ([Mar83]). RAPID and ALDS use full transposition.

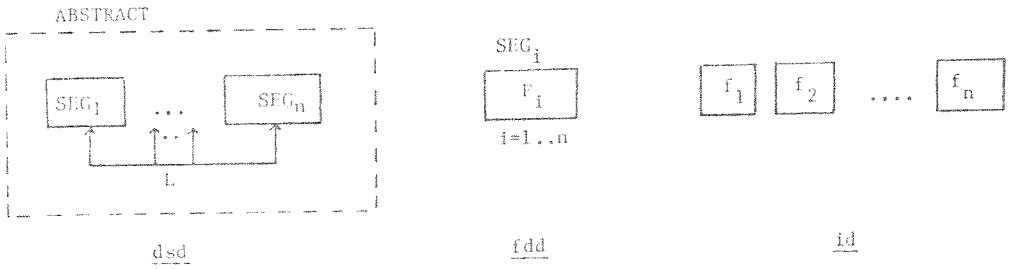


Figure 14. Full Transposition Transformation

The address converter transformation treats all fields of an abstract record as an indivisible unit of data and segments it from the abstract record. What results is an ADDRESS_CONVERTER record and a CONCRETE record connected by linkset L (see Fig. 15). The ADDRESS_CONVERTER record contains only the field P_L; the CONCRETE record contains all the fields of the abstract record plus field C_L. (Normally L is a singular pointer with parent pointers as shown in Fig. 15.id, but there are variations).

As we shall see later in Section 3, it is common for pointers to reference abstract records. The goal of the address converter 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 ADDRESS_CONVERTER record and updating the P_L pointer each time its CONCRETE record moves. ADABAS, INGRES, and DMS-1100 utilize the address converter transformation.

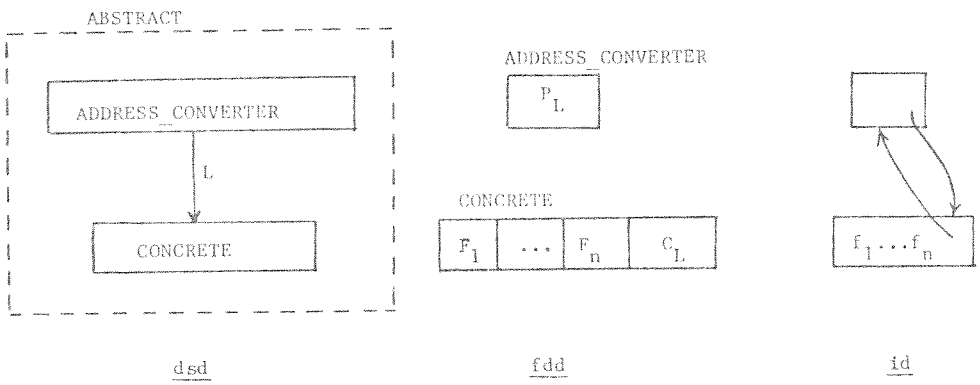


Figure 15. Address Converter Transformation

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

Division may occur with or without duplication of fields. Figure 16 shows the result of applying division without duplication to the abstract record of Figure 6. Figure 17 shows the division of the same abstract record with the duplication of field F_1 in each segment. (Note that $:F_2 \ldots F_n$ denotes a segment of the string of fields $F_2 \ldots F_n$). As additional examples, Figures 3 and 4 show how SYSTEM 2000 and MRS divide the inverted list field of the abstract index record of Figure 1. Figure 2 shows how IMS and RAPID divide the inverted list field into segments containing an individual pointer and duplicate the data value field in each segment.

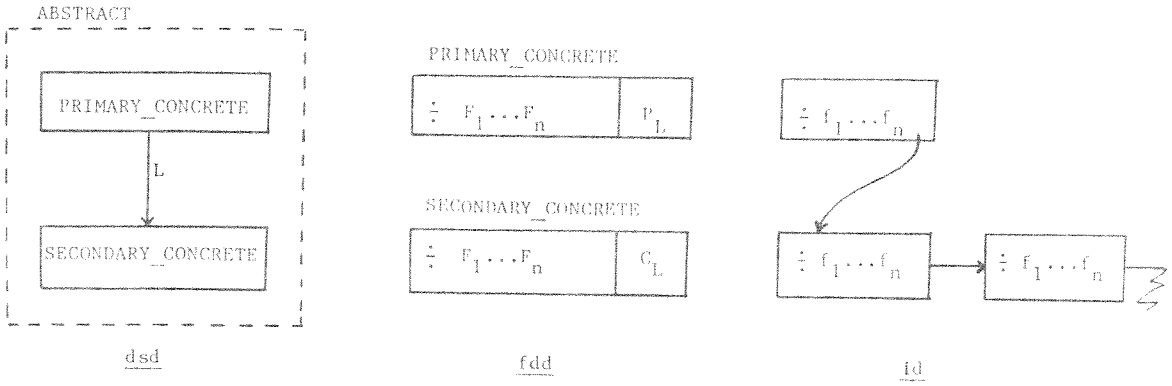


Figure 16. Division without Duplication.

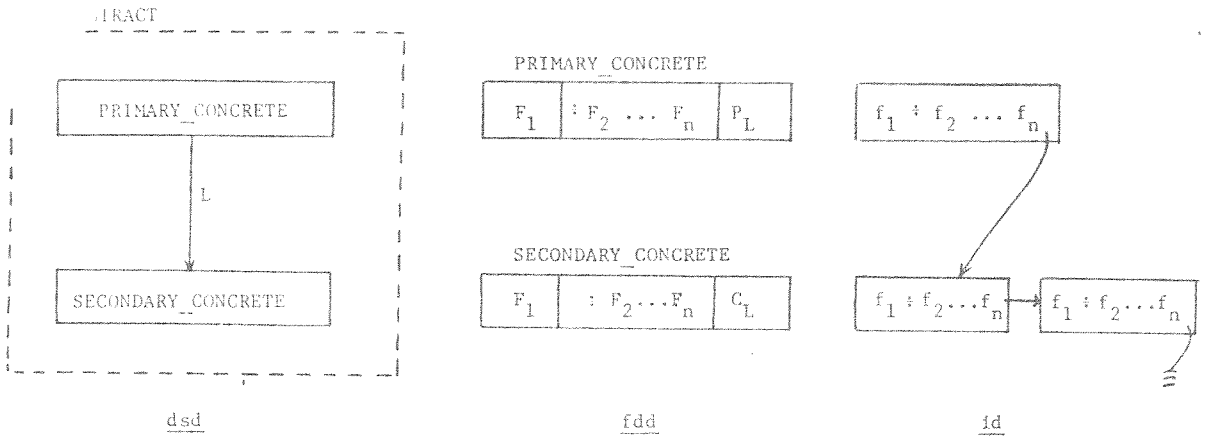


Figure 17. Division with Duplication

Actualization. Most DBMSs have some facility to relate logical record types. DMS-1100 allows sets to be defined; ADABAS allows couplings. The materialization of a logical relationship as one or more linksets is called actualization. (That is, actualization is the means by which logical relationships are mapped to physical structures). Figure 18 illustrates a relationship between logical record types F and G that is actualized by a pointer array linkset.

Actualization can be with or without field duplication. Normally it is without. ADABAS, SYSTEM 2000, and DMS-1100 use actualization without duplication. 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 fields), the fields that are copied may contain single data values or they may have a variable number of values. IMS and TOTAL use actualization with duplication.[4]

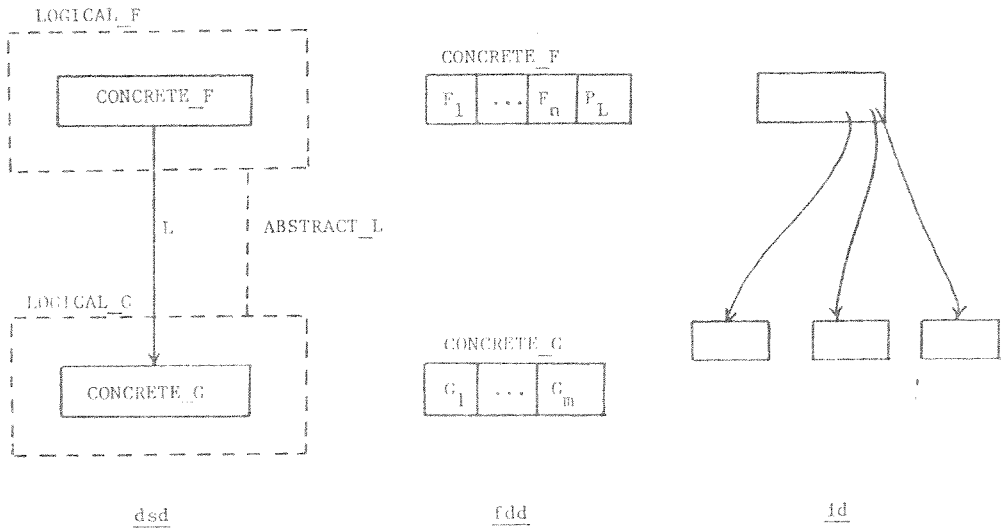


Figure 18. Actualization of Logical Relationships

[4] It is worth noting that the idea of actualization was considered some time ago in a rather different context. Mitoma, Berelian, and Irani ([Mit75], [Ber77]) addressed a DBTG database design problem. Their approach was to start with a binary data model of the database. By iteratively applying (what we call) actualization transformations, a DBTG schema was produced.

Layering. Physical database architectures can be multilayered, where each layer has well-defined notions of concrete records, file structures, and secondary-storage data blocks. A data block and its address on one layer correspond to an abstract record on the next lower layer (Fig. 19). IMS and RAPID explicitly use layering.

Layering is occasionally implicit. UNIX, for example, provides the abstract view of secondary storage as a sequential sequence of bytes. DBMSs such as INGRES and MRS define sequences of 2048 or 512 bytes as "blocks" and use these blocks to build unordered, B+ tree, and indexed-sequential file structures. In reality, UNIX treats such "blocks" as fixed length records and stores them on disk using the standard UNIX file structure ([Rit74]).

Figure 19. Layering.

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. If these conditions are not met, the abstract record is treated as a concrete record. The transform used to model these situations is the "null" transformation (Fig. 20). Models of the physical structures of SYSTEM 2000 and INQUIRE utilize the null transform.

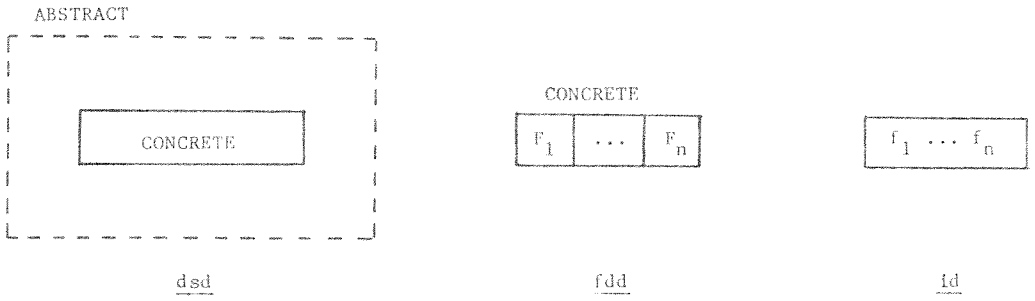


Figure 20. Null Transformation.

It is believed that these nine transformations are sufficient to derive the concrete record types of most commercial and specialized database systems. Since only a small number of DBMSs have so far been examined, it is possible that other transformations may exist. However, this question should be resolved satisfactorily by the examination of the underlying structures of other DBMSs.

In the following section, we will outline a general methodology for modeling a database management system using these transformations.

3. Modeling Methodology

The approach to model the physical structures of a DBMS begins with a data structure diagram (dsd) which captures the different kinds of logical relationships that might exist between logical record types. For example, relationships between logical record types in SYSTEM 2000 are strictly hierarchical; the starting dsd for SYSTEM 2000 would be a hierarchy of record types. (Note that as a general rule, starting dsds are not unique. See Appendices I and III for examples). DMS-1100 and ADABAS support nonhierarchical relationships. A network of record types would be used here. Relational databases, if cast into a dsd model, would also begin with a network of logical record types. Some file management systems, such as RAPID and ALDS, do not explicitly support relationships between record types. In these cases, the starting dsd would be a single logical type.

The next step is to actualize all logical relationships. This introduces parent and child fields to the record types that are related by linksets. To distinguish logical records from those that contain parent and child fields, we will call records of the latter type as abstract logical records. By actualizing all logical relationships, it should be possible to define a single "generic" abstract logical type which embodies the different abstract logical record formats which could be encountered. (Examples of which can be found in the derivations of ADABAS and SYSTEM 2000 in Appendices I and III).

There are some DBMSs that actualize logical relationships solely by join operations. As a consequence, no parent and child fields are introduced by actualization. In such cases, it is simpler to skip the above-mentioned steps and begin with a single logical record type. (We do this in the derivations of INQUIRE and INGRES in Section 4 and Appendix II).

Once the starting point (e.g., a logical or generic abstract logical

record type) has been determined, the derivation proceeds in well defined steps, where one or more elementary transformations may constitute a single step. A step is normally identified with all transformations that are applied to a single record type. The sequence of transformations that comprise a derivation follows an intuitively evident course where abstract records are progressively made more and more concrete. This progression can be seen in any of the derivations presented in this paper.

The result of applying elementary transformations to logical record types is a set of concrete record types and their linkset interconnections. Following the idea of decomposition ([Bat81]), the implementation of the physical database is known when the implementation of each linkset and concrete record type is known. Linkset implementations are specified whenever a linkset is introduced in a derivation. As for the storage structures of concrete records, a derivation terminates with a declaration of what file structures are used to store the instances of each concrete record type. It is here that blocking factors, primary keys, overflow methods, file placement, etc. are given.

One final note concerns the representation of logical records. We will find it convenient to view a logical record simply as a collection 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; it might also involve the use of special data structures (e.g., pointers or count bytes) to separate the contents of repeating or variable length fields (see [Max73]). The actual encoding that a DBMS uses to input and output its records is irrelevant to our derivations. That is, the physical structures of a DBMS are not dependent on the input/output encoding of logical records. For this reason, we shall ignore such encodings.

In the following section and in the Appendices, we apply this methodology to model the physical structures of INQUIRE, ADABAS, SYSTEM 2000, and INGRES.

4. The Physical Architecture of INQUIRE

INQUIRE is a product of Infodata Systems Inc. Over 300 installations in North America and Europe are using it. INQUIRE creates a distinct physical database for each logical record type defined by a user. Interconnections between logical record types are implicit; they are realized by join operations rather than by physical structures. The underlying structures used by INQUIRE, therefore, can be understood by examining how records of a single logical type are stored.

The generic LOGICAL record type supported by INQUIRE is shown in Figure I1. 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. In general, LOGICAL records have variable lengths.

LOGICAL record types are the record types that are defined at the schema level; LOGICAL records are the records that are processed by INQUIRE users.

Figure I1. Generic Logical Record of INQUIRE

The concrete record types of INQUIRE are derived in the following way. First, INQUIRE augments a delete flag DF to every LOGICAL record. This flag is used to mark deleted LOGICAL records. Next, INQUIRE allows scalar and repeating fields to be indexed. Field F_j is indexed by extracting it from LOGICAL records. An ABSTRACT_INDEX$_j$ record type is produced in the process (Fig. I2). (The notation ( )... in Figure I2.dsd means that an extraction of different fields may occur zero or more times. For each data field that is extracted (i.e., indexed), there is a child field in the ABSTRACT_LOGICAL type. The child fields $C_{I_{j_1}}$, $C_{I_{j_2}}$, ... in Figure I2.fdd are the result of extracting fields F_{j_1}, F_{j_2}, ...). All fields are indexed in this manner.

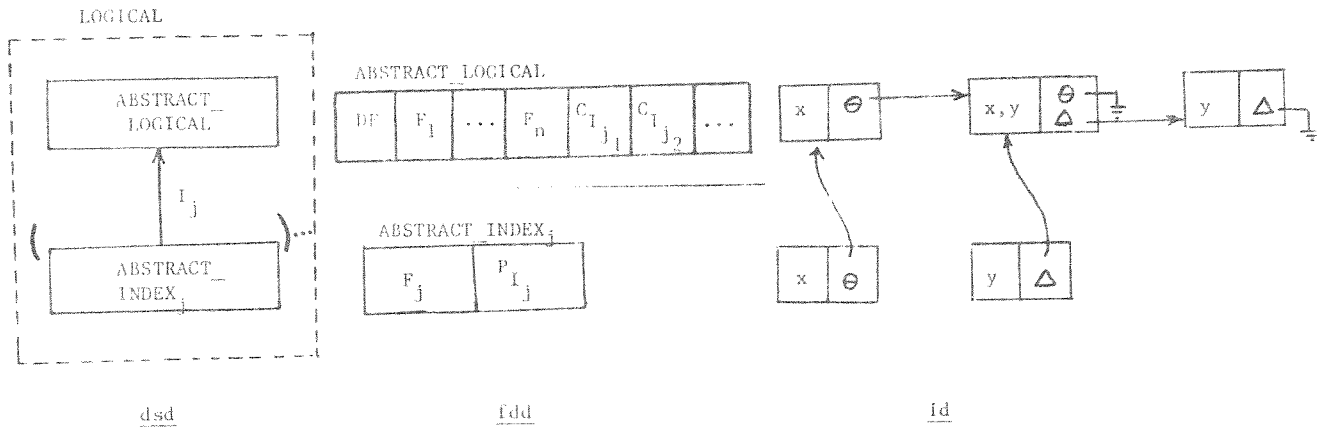


Figure I2. Extraction of LOGICAL Records

Linkset I_j, which connects ABSTRACT_INDEX$_j$ to ABSTRACT_LOGICAL, is an M:N multilist with count fields. Records are linked in descending (physical) address order. A conventional (or 1:N) multilist, as defined by the UM and elsewhere, links a parent record to its child records by a linear list. Each child record is connected to at most one parent record. An M:N multilist, in

contrast, allows child records to have more than one parent record.[5] This is accomplished by assigning a distinct fixed-length binary value, called a binkey, to each multilist occurrence. A binkey is stored in the parent record and is paired to each pointer of its list so that pointers of one multilist can be distinguished from those of another. In Figure I2.id, the binkey for the multilist occurrence with x as its parent record is ⊙ and for the one with parent record y it is Δ.[6] M:N multilists arise because an indexed field may be repeating (i.e., an indexed field may contain several distinct data values).[7]

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".

Consider, for example, the retrieval of all records of a LOGICAL 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 physical structures of INQUIRE. The ABSTRACT_INDEX records for field F_j are

---

5 So that there is no ambiguity about the distinction between 1:N and M:N linksets, 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 any one time.

6 It is envisioned with a generalized UM that it will be sufficient to say that a linkset is an M:N multilist, M:N pointer array, etc., without resorting to a detailed description of its realization, as we have done here.

7 Note that in field P_{I_j} there is a subfield containing the number of records on a list. This field was not shown in Figure I2.id, but if it were, both ABSTRACT_INDEXj records would contain the value 2.

made concrete by augmenting the characteristic string "F_j=" (i.e., the field name followed by an equals) to each data value. This is done only to prefix fields (Fig. I3). No augmentation is performed on simple fields (Fig. I4).

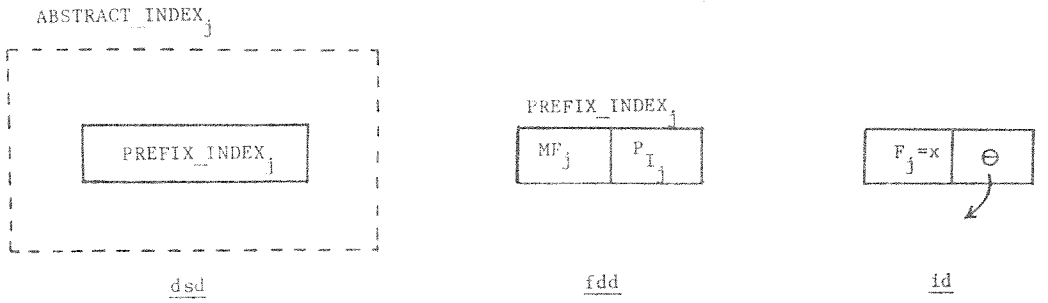


Figure I3. Augmentation of ABSTRACT_INDEX$_j$ Records.

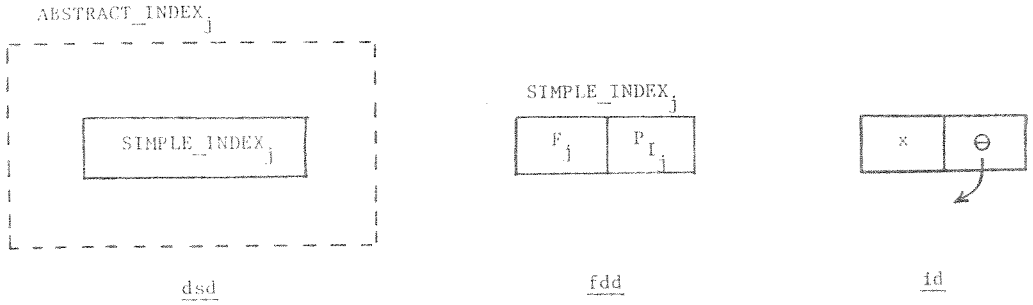


Figure I4. Null Transformation of ABSTRACT_INDEX$_j$ Records.

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.

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

An ABSTRACT_LOGICAL record of Figure I2 is materialized by collectively segmenting all C_{I_j} fields from the data fields $F_1 \cdots F_n$ (see Fig. I5). The delete flag DF is duplicated in both segments. This segmentation produces the ABSTRACT_SEARCH and ABSTRACT_DATA record types. Linkset D, which connects ABSTRACT_SEARCH to ABSTRACT_DATA, is a singular pointer with parent pointers. (A singular pointer is a pointer array that contains precisely one pointer). Figure I5.id shows the result of this segmentation on the ABSTRACT_LOGICAL records of Figure I2.id.

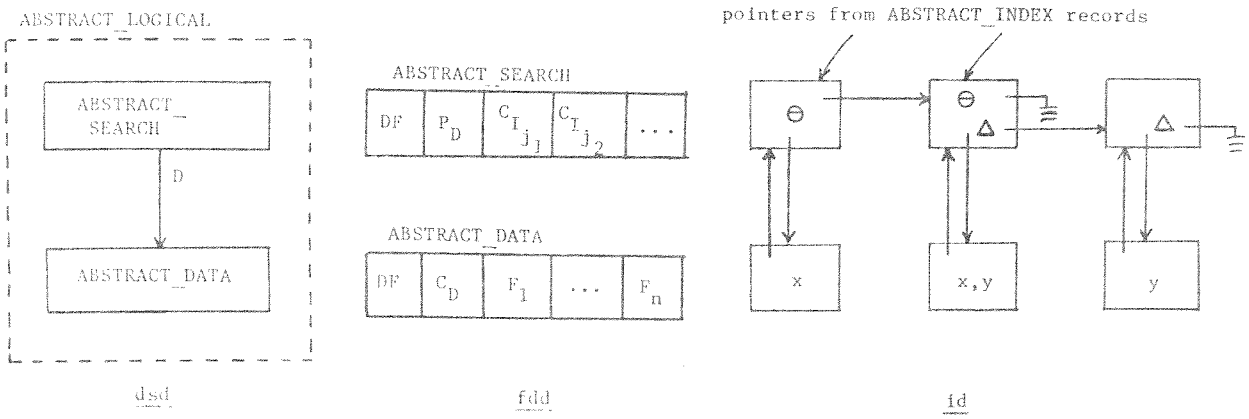


Figure I5. Segmentation of ABSTRACT_LOGICAL Records.

ABSTRACT_SEARCH records are variable in 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 and maintaining variable length records, INQUIRE divides the collection of all C_{I_j} fields into fixed length segments. The primary segment, which contains the P_D field of ABSTRACT_SEARCH, is the SEARCH record type; all secondary segments are instances of the SEARCH_OVERFLOW record type (Fig. I6). Note that the delete flag DF is duplicated in the primary and secondary segments. SEARCH and SEARCH_OVERFLOW records are connected by linkset S which is a 1:N multilist with parent pointers (Fig. I6.id). SEARCH_OVERFLOW records are linked in order of ascending physical addresses.

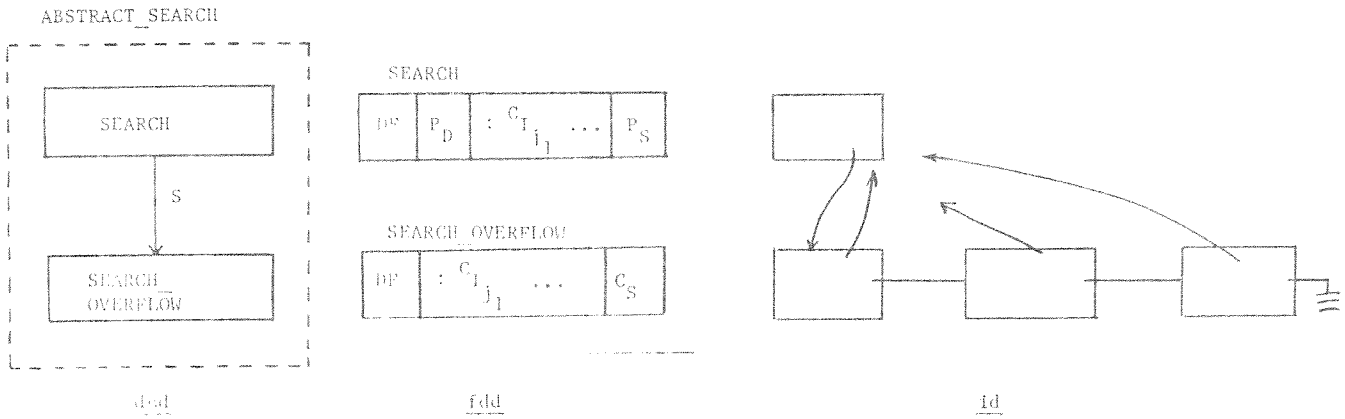


Figure I6. Division of ABSTRACT_SEARCH Records.

The ABSTRACT_DATA record of Figure I5 is materialized in two steps (see Fig. I7). First, all of its instances are collected together on a single list. Linkset R, which realizes the collection, is a 1:N multilist (with precisely one occurrence). Records are linked in reverse chronological order. Second, instances of ABSTRACT_DATA are usually variable in length because some fields are repeating. INQUIRE materializes an ABSTRACT_DATA record by dividing it into a primary segment and zero or more secondary segments.[8] The primary segment (PRIMARY_SEG) is coupled to zero or more secondary segments (SECONDARY_SEG) by linkset C, which is a 1:N doubly linked multilist. SECONDARY_SEG records are linked in ascending physical address order. Note that linkset R collects together all instances of PRIMARY_SEG. Figure I7.id shows two ABSTRACT_DATA records; one is in three segments (one primary, two secondary), the other is in four segments.

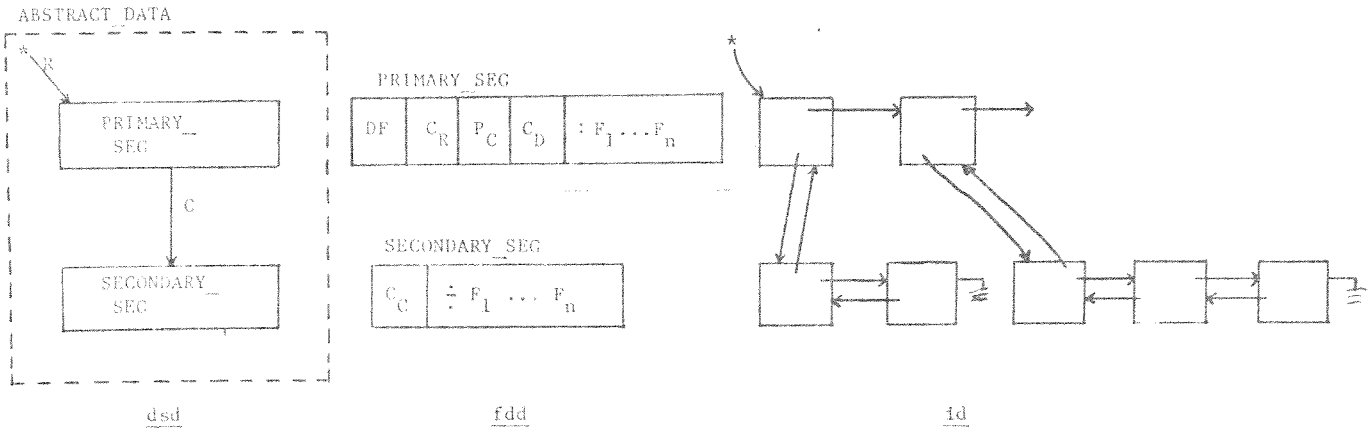


Figure I7. Division of ABSTRACT_DATA Records.

[8] Primary and secondary segments are variable in length. The length of a primary segment 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 segment. It is at this point when INQUIRE divides the ABSTRACT_DATA record.

The purpose of linkset R is now clear. In order to retrieve all LOGICAL records, linkset R must be traversed. For each PRIMARY_SEG that is encountered, all of its SECONDARY_SEG records are retrieved via linkset C. Adjoining the PRIMARY_SEG and its SECONDARY_SEG records materializes a LOGICAL record. In this way, INQUIRE realizes a scan of a LOGICAL file.

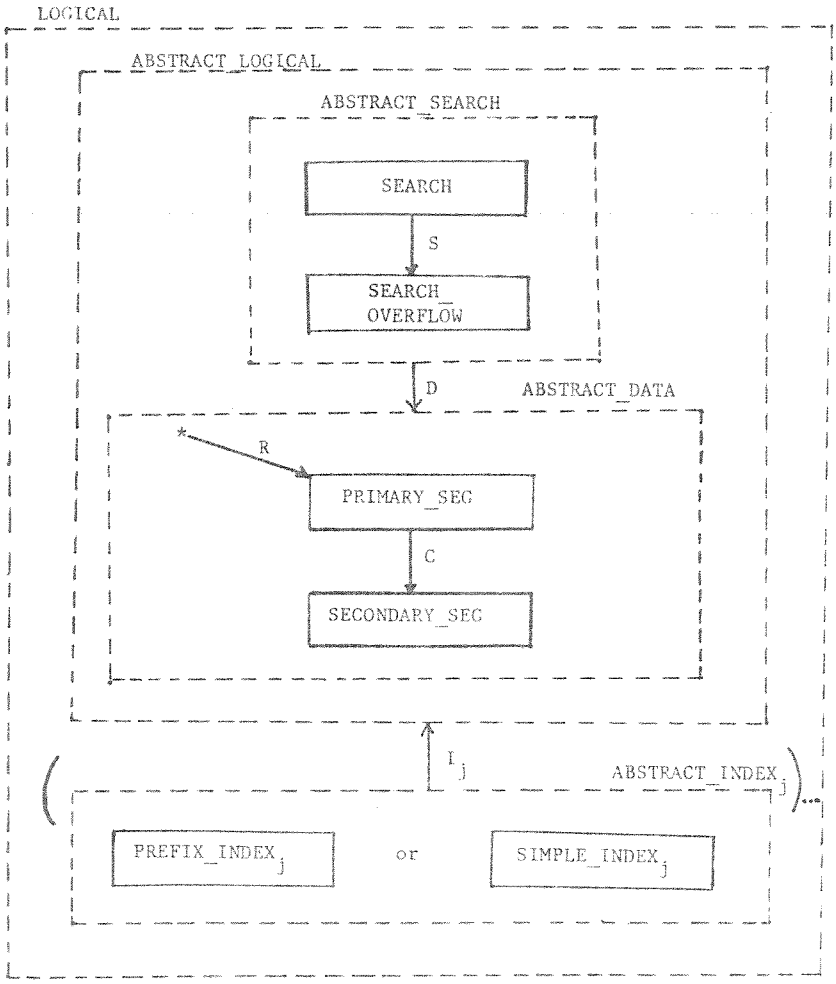
The concrete record types of INQUIRE are SIMPLE_INDEX, PREFIX_INDEX, SEARCH, SEARCH_OVERFLOW, PRIMARY_SEG, and SECONDARY_SEG. SIMPLE_INDEX and PREFIX_INDEX records are organized by a single VSAM or ISAM file structure. SEARCH and SEARCH_OVERFLOW records are organized by separate BDAM or RSDS file structures. All PRIMARY_SEG and SECONDARY_SEG records are organized by a single BDAM or RSDS file structure.[9] These four file structures are respectively called the INDEX, SEARCH, SEARCH OVERFLOW, and the DATA files in INQUIRE documention.

This ends the derivation of the physical architecture of INQUIRE. It is worth noting that our model of INQUIRE is quite accurate; the concrete record types that were derived can explain the presence and purpose of every pointer and every byte of the stored records that are documented in the INQUIRE manuals. A data structure diagram that summarizes the derivation is shown in Figure I8. Source materials used in the derivation are [Inf79a,b] and [Des83].

Finally, it is worth noting that INQUIRE, like most other DBMSs, has some support files which were not considered in this derivation. In particular, these are the DECODE, ACCOUNTING, and MACRO LIBRARY files, among others. These files could have been included, in principle, without much difficulty. But since their presence is optional and they do not constitute the core of INQUIRE's physical architecture, we omitted them for simplicity.

---

[9] In the terminology of the UM, 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.



Concrete Record or Linkset	Implementation
All PREFIX_INDEX$_j$ and SIMPLE_INDEX$_j$ occurrences	B+ tree
SEARCH	unordered
SEARCH_OVERFLOW	unordered
All PRIMARY_SEG and SECONDARY_SEG occurrences	unordered
All I$_j$ types	M:N multilist
S	1:N multilist with parent pointers
D	singular pointer with parent pointers
R	1:N multilist
C	1:N doubly linked multilist

Figure 18. The Physical Architecture of INQUIRE

5. Future Work and a Perspective on Functional Equivalence

Understanding the physical structures of a DBMS is a necessary precondition to understand the DBMS's behavior and performance. But it is not sufficient. Operations on these structures must also be considered and performance models need to be developed. We will address each in turn.

Modeling operations on physical databases will require the use of abstract data types. This can be seen by noting that each step of a functional equivalence derivation defines a new record type. Operations on logical record types are realized by operations on abstract logical types, and these operations are in turn realized by operations on less-abstract types, and so on. The programming languages notion of a module - i.e., the idea of encapsulating a record type with all operations that can be performed on it ([Lis77], [Ghe82]) - embraces the idea of functional equivalence and provides a natural way to model operations.

Following this approach, we have also found that it leads to a comprehendible methodology for designing physical database architectures. Although the value of abstract data types in database implementations has long been recognized ([Ham76], [Row79], [Bar81]), the mechanism by which modular design concepts are applied at the "internal" or physical level has not been well understood. We feel that functional equivalence can improve and clarify these methodologies. Results on this topic and that of modeling operations in general are presented in forthcoming works ([Wis83], [Bat84]).

Performance and design packages for commercial DBMSs can be developed once it is known how operations are mapped from the logical to the physical levels. The development of these packages will require the integration of performance prediction techniques with the descriptive techniques of

functional equivalence. Presently we feel that the integration process will be straightforward. This does not mean that optimization problems for database design will be any easier to solve; it simply means that the results of an optimization will be tailored to the peculiarities of a specific DBMS. For example, it is well known that papers on index selection have used optimization models that were not tied to any existing database system. Thus there is reason to believe that the results of index selection for an INGRES database may be different (albeit slightly) from that of an ADABAS or INQUIRE database. Research on such topics should prove to be quite interesting.

The immediate usefulness of functional equivalence is seen in three ways. First, our work may be the start of a comprehensive reference to the architectures of operational DBMSs. Database systems are being developed today and will be developed many years from now. Accurate descriptions of the architectures of commercially successful DBMSs should be quite useful to future DBMS designers. Second, functional equivalence derivations provide a useful medium of communication. In just a few pages, the physical structures of a DBMS can be conveyed with considerable detail and precision. Presently, this is accomplished by reading cryptic (and often confusing) documentation and enormous software manuals; all of which are quite time consuming. Third, we feel that knowledge of the physical architectures of well-known DBMSs ultimately improves ones understanding of physical databases in general.

Presently, we are surveying the architectures of other DBMSs. We do so to further demonstrate the power of functional equivalence, but also to discover what simple files (e.g., multirecord type file structures, B+ tries) and linksets (M:N multilists, record clustering) still need to be encorporated into existing general models of physical databases. In addition, these surveys will help uncover any remaining elementary transformations which have

not yet been noticed. It is believed all of the major transformations have been discovered, but no theoretical result can be stated at this time to support this claim.

Finally, we noted in Section 2 that a tenth transformation exists. It is commonly referred to as horizontal partitioning ([Alsb75], [Cer83]). The basic idea is to partition a file of records into two or more groups. Differential files ([Sev76], [Agh82]), for example, partition records into two groups: modified and unmodified. Distributed databases may support the distribution of records of a single file at different sites. Unlike other elementary transformations, no explicit physical structures (e.g., delete flags, pointers, etc.) are added to horizontally partitioned records to interrelate their groups. However, metadata must be present (somewhere) in DBMSs to make such relationships explicit. Thus there appear to be tranformations that introduce structure at the "metadata" level, not at the abstract and concrete data record levels. Further research on horizontal partitioning is necessary.

6. Conclusions

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

It has been shown that functional equivalence provides the necessary means to model the complex structures of well-known DBMSs in a simple, concise, systematic, and comprehendible way. This is the primary contribution of this paper. But we have also explained the relationship of functional equivalence to abstract data types and its role in DBMS design methodologies and in the development of performance packages for commercial DBMSs. We believe functional equivalence is an important step toward tieing physical database theory to practice.

Acknowledgements. I gratefully acknowledge the encouragement, support, and ideas I received from the following people: Ignacio Casas of the University of Toronto; Jim Desper of Infodata Systems, Inc.; John McCarthy of Lawrence Berkeley Laboratories; Alan Wolfson of Software AG; Paul Butterworth of Relational Technology; and Tim Wise, Stan Su, Sham Navathe, and Jim Parkes of the University of Florida.

References

[Agh82] H. Aghili and D.G. Severance, "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.

[Als75] P.A. Alsberg, "Space and Time Savings Through Large Database Compression and Dynamic Restructuring", Proc. IEEE 63,8 (Aug. 1975), 1114-1122.

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

[Bar81] A.J. Baroody and D.J. DeWitt, "An Object-Oriented Approach to Database System Implementation", ACM Trans. Database Syst. 6,4 (Dec. 1982), 576-601.

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

[Bat84] D.S. Batory and T. Wise, "A Methodology for Modeling and Designing the Internal Level of a Database Management System", to appear.

[Ber77] E. Berelian and K.B. Irani, "Evaluation and Optimization", Proc. VLDB 1977, 545-555.

[Bur81] R.A. Burnett, "A Self-Describing Data File Structure for Large Data Sets", in Computer Science and Statistics: Proc. of the 13th Symposium on the Interface, Springer-Verlag, New York, 1981, 359-362.

[But83] P. Butterworth, (Relational Technology, Inc.), Technical Discussion, 1983.

[Cas81] I. Casas-Roposo, "Analytic Modeling of Dtabase Systems: The Design of a SYSTEM 2000 Performance Predictor", M.Sc. Thesis, Dept. of Computer Science, University of Toronto, 1981.

[Cas82] I. Casas-Roposo, Technical discussion, 1982.

[Car81] J.V. Carlis and S.T. March, "A Computer-Aided Data Base Design Methodology", Auerbach Report on Design and Development, TR 23-01-11, 1981.

[Cer83] S. Ceri, M. Negri, and G. Pelagatti, "Horizontal Partitioning in Database Design", Proc. ACM SIGMOD 1982, 128-136.

[Cin79] Cincom Systems, Inc., "TOTAL PDP-11 Programmers Reference Manual", Cincinnati, Ohio, 1979.

[Cul81] Cullinane Database Systems, Inc., "IDMS System Overview", Westwood, Massachusetts, 1981.

[Dat82] C.J. Date, An Introduction to Database Systems, 3rd edition, Addison-Wesley, 1982.

[Des83] J. Desper, (Infodata Systems, Inc.), Technical Discussion, 1983.

[Fre60] E. Fredkin, "Trie Memory", Comm. ACM 3 (1960), 490-500.

[Fry74] J.P. Fry and D. Jeris, "Towards a Formulation of Data Reorganization", 1974 ACM SIGMOD Workshop on Data Description, Access and Control, 77-107.

[Ges76] Gesellshaft fur Mathematik und Datenverarbeitung, "ADABAS: Database Systems Investigation Report, Vol. 2, Part 1", Istitute fur Informationssysteme, Bonn, West Germany, 1976.

[Ghe82] C. Ghezzi and M. Jazayeri, Programming Language Concepts, John Wiley and Sons, Inc, New York, 1982.

[Ham76] M. Hammer, "Data Abstractions for Databases", Proc. Conf. Data: Abstractions, Definition, and Structure, SIGPLAN Notices 11 (Special Issue), 1976, 58-59.

[Hel75] G.D. Held and M.R. Stonebraker, "Storage Structures and Access Methods in the Relational Database Management System INGRES", Proc. ACM Pacific 1975, 26-33.

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

[Hsi70] D. Hsiao and F. Harary, "A Formal System for Information Retrieval from Files", Comm. ACM 13,2 (Feb. 1970), 67-73.

[Huf52] D.A. Huffman, "A Method for the Construction of Minimal Redundancy Codes", Proc. IRE 40 (Sept. 1952), 1098-1101.

[IBM81] IBM, "IMS/VS Version 1: Data Base Administration Guide", San Jose, California, 1981.

[Inf78] Informatics, Inc., "SHRINK/2 Users Guide", Canoga Park, California, 1978.

[Inf79a] Infodata Systems, Inc., "INQUIRE Basic Training Course", Pittsford, New York, 1979.

[Inf79b] Infodata Systems, Inc., "INQUIRE Database Design and Loading Manual", Pittsford, New York, 1979.

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

[Kro77] D. Kroenke, Database Processing, S.R.A. Inc., Chicago, 1977.

[Lem79] A. Lempel, "Cryptology in Transition", ACM Comp. Surveys 11,4 (Dec. 1979), 285-305.

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

[Nav76] S.B. Navathe and J.P. Fry, "Restructuring for Large Databases: Three Levels of Abstraction", ACM Trans. Database Syst. 1,2 (June 1976), 138-158.

[Nat77] National Bureau of Standards, Federal Information Processing Standards, Publication 46, 1977.

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

[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.

[Mar83] S.T. March, "Techniques for Structuring Database Records", to appear in ACM Computing Surveys, March 1983.

[Max73] W.L. Maxwell and D.G. Severance, "Comparison of Alternatives in an Information System", Proc. Wharton Conf. Research on Computers in Organizations, University of Pennsylvania, Philadelphia, (Oct. 1973), 1-16.

[Mit75] M.F. Mitoma and K.B. Irani, "Automatic Data Base Schema Design and Optimization", Proc. VLDB 1975, 286-321.

[Row79] L.A. Rowe and K.A. Shoens, "Data Abstractions, View, and Updates in RIGEL", Proc. ACM SIGMOD 1979, 71-81.

[Rit74] D.M. Ritchie and F. Thompson, "The UNIX Time-Sharing System", Comm. ACM 17,7 (July 1974), 365-375.

[Sen73] M.E. Senko, E.B. Altman, M.M. Astrahan, P.L. Fehder, "Data Structures and Accessing in Database Systems", IBM Syst. Jour. 12,1 (1973), 30-93.

[Sev72] D.G. Severance, "Some Generalized Modeling Structures for use in Design of File Organizations", Ph.D. Thesis, University of Michigan, Ann Arbor, Michigan, 1972.

[Sev76] D.G. Severance and G.M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", ACM Trans. Database Syst. (Sept. 1976), 256-267.

[Shn82] B. Shneiderman and G. Thomas, "An Architecture for Automatic Relational Database System Conversion", ACM Trans. Database Syst. 7,2 (June 1982), 235-257.

[Sof77] Software AG of North America, Inc., "ADABAS: Introduction", Reston, Virginia, 1977.

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

[Sta73] Stanford University, "Design of SPIRES: Vol. I and II", Center for Information Processing, Stanford University, 1973.

[Sta81] Statistics Canada, "RAPID Internals Manual", Ottawa, Ontario, Canada, 1981.

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

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

[Teo82] T.J. Teorey and J.P. Fry, Design of Database Structures, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[Tsi77a] D.C. Tsichritzis and F. Lochovsky, Data Base Management Systems, Academic Press, New York, 1977.

[Tsi77b] D.C. Tsichritzis and A. Klug (eds.), "The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems", Information Syst. 3, 1978, 173-191.

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

[Wel72] M. Wells, "File Compression using Variable Length Encodings", Computer J. 15,4 (1972), 308-313.

[Wis83] T. Wise, M.Sc. Thesis, Dept. of Computer and Information Sciences, University of Florida, to appear.

[Wol82] A. Wolfson, (Software AG of North America, Inc.), Technical discussion, 1983.

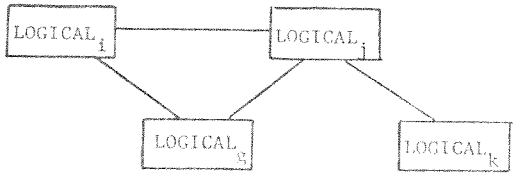
[Won76] E. Wong and K. Youseffi, "Decomposition - A Strategy for Query Processing", ACM Trans. Database Syst. 1,3 (Sept. 1976), 223-241.

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

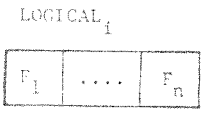
Appendix I. ADABAS

ADABAS is a product of Software AG, Inc. A typical ADABAS database is populated with one or more logical record types which may be related either explicitly by couplings or implicitly by join operations. A representative ADABAS data structure diagram is shown in Figure A1. Couplings are represented by lines that connect two different logical record types. ADABAS does not allow for a record type to be coupled with itself, or for more than one coupling to exist between two record types at any one time.[1]

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



dsd



fdd

Figure A1. A Representative ADABAS dsd

---

[1] Couplings are used in only 1-2 percent of ADABAS databases because their utility is limited to processing specialized queries and they degrade performance significantly for update-intensive files ([Ges76]). Couplings are supported in the most recent release of ADABAS, but their use is not recommended. Join operations are promoted instead.

When record types are coupled, a connection between individual records is made on the basis that they share a common value in designated fields. Because fields may be repeating, couplings can be M:N. Figure A2.id illustrates an M:N coupling.

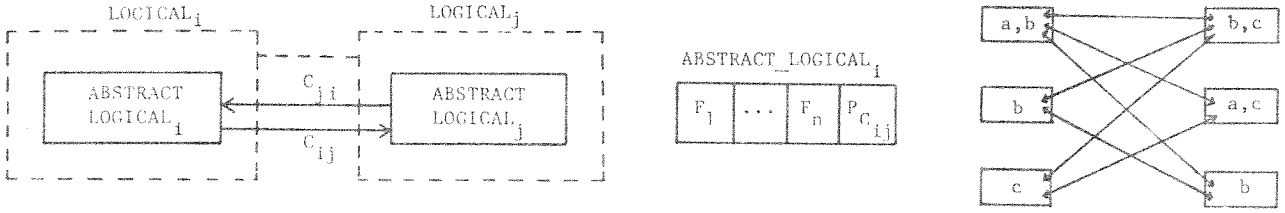


Figure A2. Actualization of a Coupling

The concrete record types of ADABAS are derived in the following way. Each coupling is actualized by a pair of linksets; each record type is the parent of one linkset and both linksets are M:N pointer arrays.[2,3] The pointers of each array are maintained in order of ascending addresses. Figure A2 shows the actualization of the coupling between the LOGICAL$_i$ and LOGICAL$_j$ record types. Two ABSTRACT_LOGICAL record types and two linksets are produced in the process. An actualization of the database in Figure A1 would produce a total of four ABSTRACT_LOGICAL record types and eight linksets.

---

2 An M:N pointer array differs from the conventional 1:N pointer array in that a child record can have more than one parent record.

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

The generic form of an ABSTRACT_LOGICAL record is shown in Figure A3. An ABSTRACT_LOGICAL record is the parent of m linksets, L_1 ... L_m, which were produced by the actualization of m couplings. A record consists of data fields F_1 ... F_n and m parent fields P_{L_1} ... P_{L_m}.

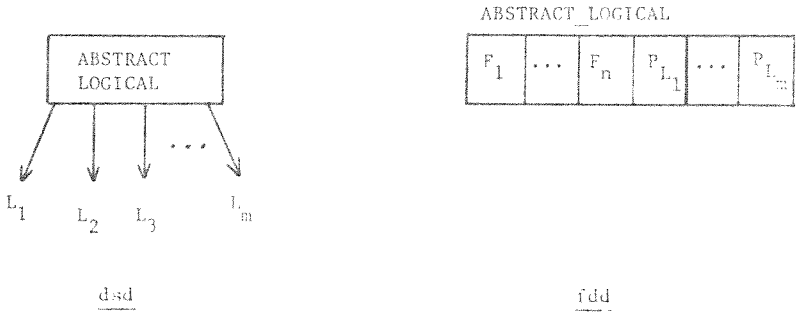


Figure A3. Generic ABSTRACT_LOGICAL Record of ADABAS

ABSTRACT_LOGICAL records are materialized in two steps (see Fig. A4). First, fields P_{L_1} ... P_{L_m} are individually segmented from the data fields F_1 ... F_n. m+1 record types result: an ABSTRACT_DATA type and an ABSTRACT_ASSOCIATOR$_k$ type for each linkset L_k. Linkset A_k, which connects ABSTRACT_ASSOCIATOR$_k$ to ABSTRACT_DATA, is a singular pointer (i.e., a pointer array that contains precisely one pointer).

Second, ADABAS allows scalar and repeating fields to be indexed. Field F_j is indexed by extracting it from ABSTRACT_DATA. This forms an ABSTRACT_INDEX$_j$ record. Linkset I_j, which connects ABSTRACT_INDEX$_j$ to ABSTRACT_DATA, is 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 LOGICAL record type was uncoupled and had no indexed fields, it would be mapped directly to an ABSTRACT_DATA record via the null transform.

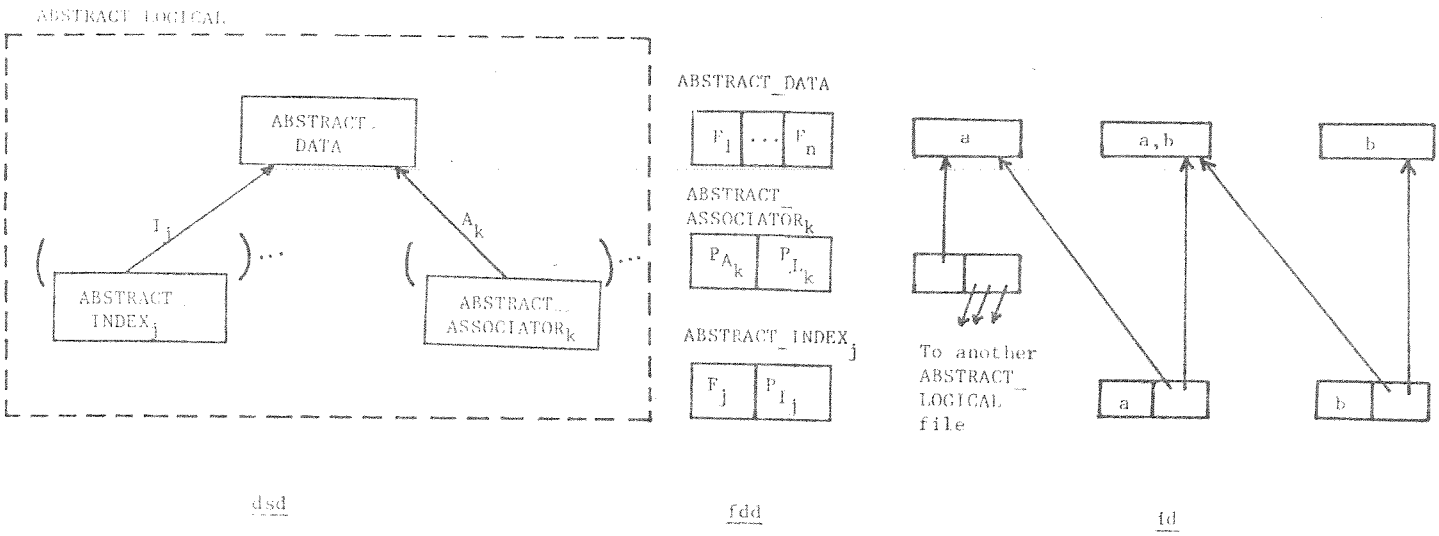


Figure A4. Segmentation and Extraction of ABSTRACT_LOGICAL Records.

Pointers to ABSTRACT_DATA records are known as internal sequence numbers (ISNs). A distinct ISN is assigned to each LOGICAL data record and is used to locate the record. Its role is further amplified later. Internal file numbers and internal field numbers, which we will collectively call IFNs, are used internally by ADABAS to reference LOGICAL record types and their constituent fields. Field numbers are distinguishable from file numbers.

An ABSTRACT_INDEX$_j$ record is materialized in three steps (see Fig. A5). First, a metadata 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 [Ges76] for more details). The encoded field is labeled F_j' in Figure A5.fdd. Third, the record may be divided into segments with the IFN and F_j' fields duplicated in each segment. (The conditions under which division occurs will be explained shortly). Figure A5.id shows how an ABSTRACT_INDEX$_j$ record with an inverted list of 100 pointers might be divided. Because of this construction, no distinction is made between the primary record (i.e., the

first segment) and secondary records (i.e., the remaining segments).[4]

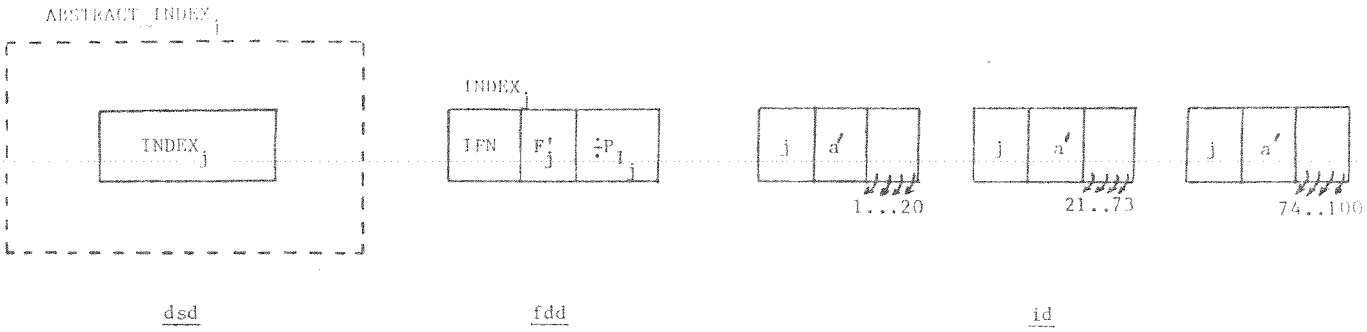


Figure A5. Augmentation, Encoding, and Division of ABSTRACT_INDEX$_j$ Records.

An ABSTRACT_ASSOCIATOR$_k$ record is materialized in a similar manner (see Fig. A6). First, a metadata field containing the IFN of the child file of linkset L_k is augmented. Second, the record may be divided into segments with the IFN and P_{A_k} fields duplicated in each segment. Figure A6.id shows how an ABSTRACT_ASSOCIATOR$_k$ record with a pointer array of 100 pointers might be divided.

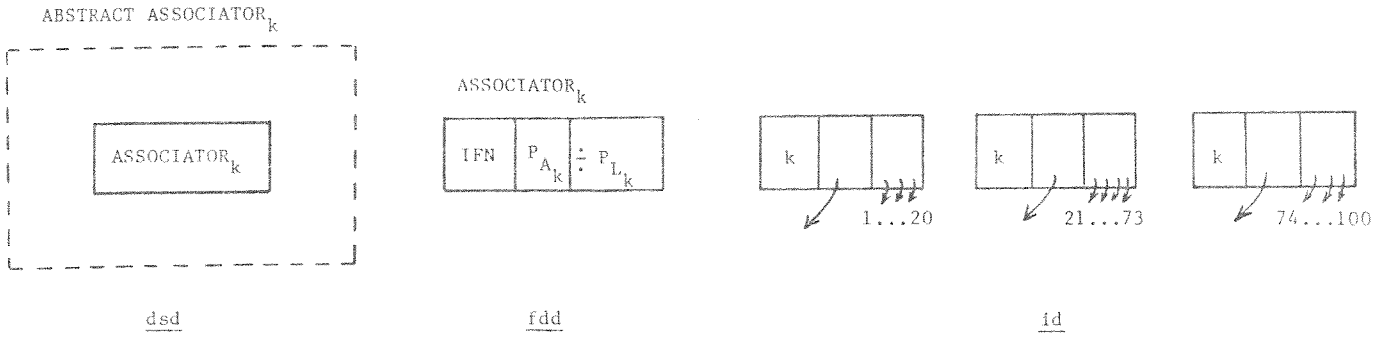


Figure A6. Augmentation and Division of ABSTRACT_ASSOCIATOR$_k$ Records.

[4] If a distinction were made or if a linkset were to be introduced, the structure connecting the primary to the secondary records would be an relational linkset with link key (IFN, F'$_j$). A relational linkset does not explicitly connect parent to child records by pointers, but rather the connection is implied by the sharing of a common key called a link key. Relational databases normally rely on relational linksets for connections between relations.

ADABAS forces concrete records of type $INDEX_j$ and $ASSOCIATOR_k$ to have a similar format so that they can be organized by a single file structure rather than having a separate file structure for each indexed field and file coupling. The file structure is a B+ trie, which is similar to B+ trees in that file growth is accommodated by node splitting.[5] The division of an $ABSTRACT\_INDEX_j$ or $ABSTRACT\_ASSOCIATOR_k$ record is a result of node splitting. When a node splits, two nodes are created; both are approximately half full. Although $INDEX_j$ and $ASSOCIATOR_k$ 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, evenly loading both nodes is not possible without dividing one record into two and storing them in different nodes. Figure A7 illustrates the splitting of a node and the division of record R3 into R3' and R3".

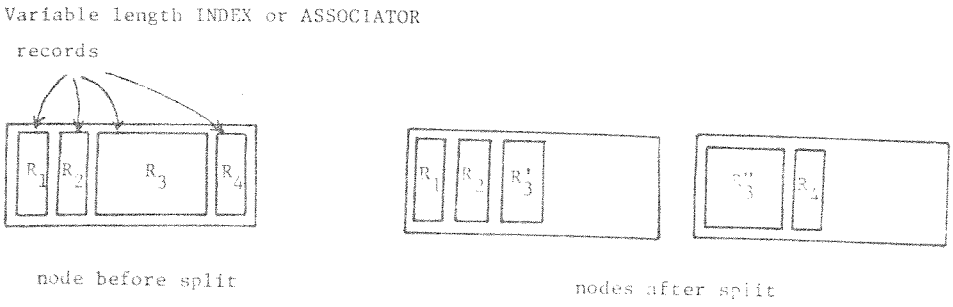


Figure A7. Illustration of Division of INDEX and ASSOCIATOR Records.

[5] A B+ trie is a hybridization of the Trie ([Fre60], [Teo82]) and B+ tree. The B+ trie used in the most recent release of ADABAS has from one to six levels. The top levels partition INDEX and ASSOCIATOR records on their IFN and F_j^i or P_{A_k} values. The second lowest level partitions records on F_j^i or P_{A_k} and ISN values. The bottom level contains the INDEX and ASSOCIATOR records.

An ABSTRACT_DATA record of Figure A4 is materialized in two steps. First, all data fields are encoded by an ADABAS compression technique. Second, the address converter transformation is applied to the compressed fields. What results is an ADDRESS_CONVERTER record and a COMPRESSED_DATA record connected by linkset 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}. Linkset AC is materialized by a pointer to the block that contains the associated COMPRESSED_DATA record, and the COMPRESSED_DATA record has a pointer to its ADDRESS_CONVERTER record (Fig. A8). This is a 1:1 cellular serial linkset with parent pointers.

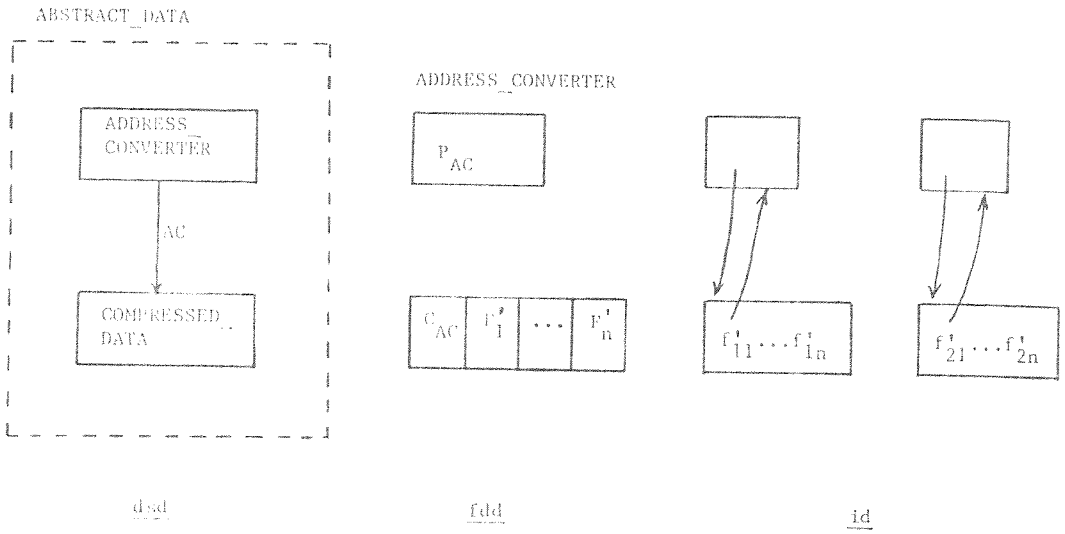


Figure A8. Segmentation and Encoding of ABSTRACT_DATA Records.

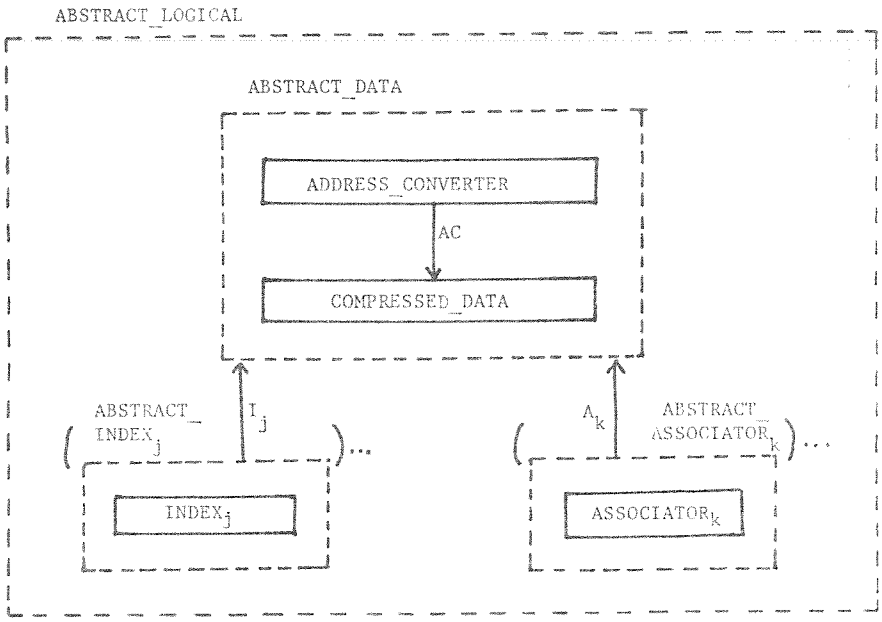
Note that ADDRESS_CONVERTER records maintain the 1:1 correspondence between ISNs and the storage locations of COMPRESSED_DATA records. Because of this correspondence, a COMPRESSED_DATA record can be relocated in secondary storage without altering the inverted lists and pointer arrays of INDEX and ASSOCIATOR 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 LOGICAL record is modified, such as adding a new value to a repeating field.

The concrete record types of ADABAS are $INDEX_j$, $ASSOCIATOR_k$, ADDRESS_CONVERTER, and COMPRESSED_DATA. Every LOGICAL file is materialized by a collection of these records and each collection is organized by a separate group of file structures. For each LOGICAL file, all occurrences of the $INDEX_j$ and $ASSOCIATOR_k$ record types are organized by a single B+ trie (see [Kro77], [Ges76], and footnote 5 of this Appendix). ADDRESS_CONVERTER and COMPRESSED_DATA records are organized by separate unordered files. An ISN is the relative location key of an ADDRESS_CONVERTER record.

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

A data structure diagram that summarizes the derivation is shown in Figure A9. Source materials used in the derivation are [Ges76], [Sof77], [Kro77], [Sof80], and [Wol82].



dsd

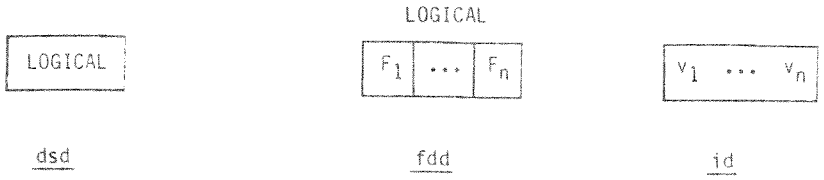
Concrete Record or Linkset	Implementation
All $INDEX_j$ and $ASSOCIATOR_k$ occurrences	B+ trie
ADDRESS_CONVERTER	unordered
COMPRESSED_DATA	unordered
All I_j types	n:m inverted list
All A_k types	singular pointer (1:1 pointer array)
AC	1:1 cellular serial with parent pointers

Figure A9. Abstract and Concrete Record Types of ADABAS.

Appendix II. INGRES

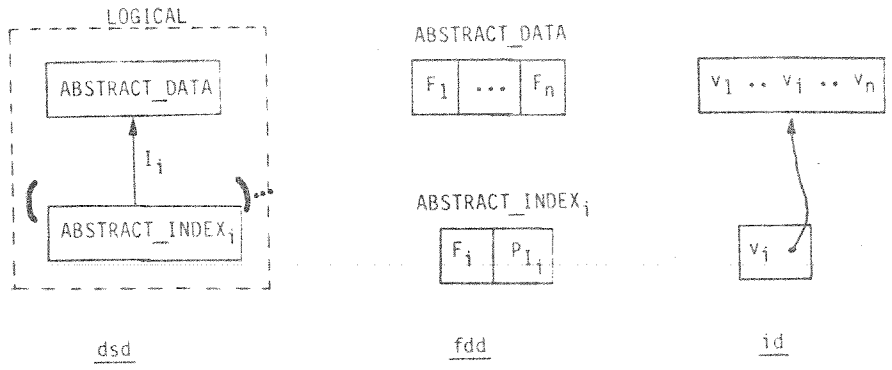
INGRES was the first major DBMS that was based on the relational model ([Hel75], [Sto76]). It was developed in the mid-1970's at the University of California, Berkeley, and is now marketed by Relational Technology, Inc.

The generic LOGICAL record type supported by INGRES consists of n scalar and elementary fields (see Fig. G1). Data values and their respective fields have fixed lengths. Relationships between two or more LOGICAL record types are realized by join operations.



G1. Generic LOGICAL Record Type of INGRES

LOGICAL records are materialized in the following way. INGRES allows elementary or compound fields to be indexed. (A compound field is defined by INGRES to be a field that consists of two to six elementary fields). Field F_i is indexed by segmenting it with duplication from LOGICAL records. This produces an ABSTRACT_INDEX$_i$ record type connected to an ABSTRACT_DATA record type by linkset I_i (see Fig. G2). I_i is a singular pointer. (The value of this pointer is called the tuple id of the LOGICAL or ABSTRACT_DATA record.) All fields are indexed in this manner. (The notation ( )... in Figure G2.dsd means that zero or more fields may be indexed.)

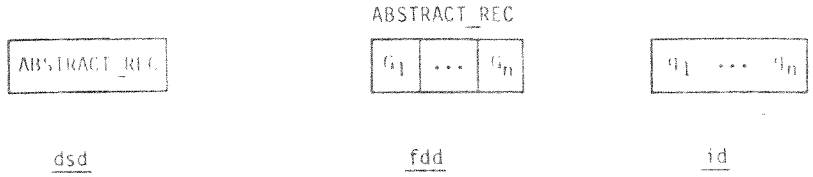


G2. Segmentation of LOGICAL Record Type

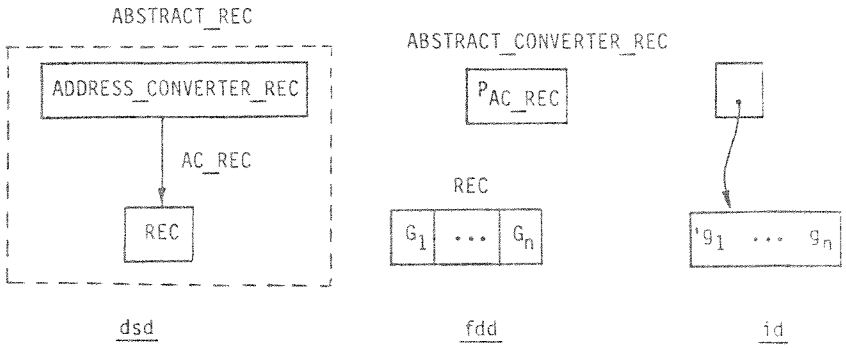
INGRES treats ABSTRACT_INDEX files as special normalized relations consisting of a data value field and a pointer field. This treatment actually simplifies the implementation of INGRES, for ABSTRACT_INDEX and ABSTRACT_DATA records are materialized by the same sequence of transformations. We define this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

Let ABSTRACT_REC consist of n fixed length fields G_1 ... G_n (see Fig. G3). ABSTRACT_REC is materialized by the address converter transformation (i.e., fields G_1 ... G_n are segmented from ABSTRACT_REC). An ADDRESS_CONVERTER_REC and REC record types connected by linkset AC_REC are produced in the process (see Fig. G4). ADDRESS_CONVERTER_REC is fixed length and contains the single field P_{AC_REC}. REC is identical to ABSTRACT_REC in terms of its contents. AC_REC is a singular pointer with special parent-child clustering properties (to be explained shortly).[1]

[1] AC_REC would be classified as a cellular-sequential singular pointer in the UM.

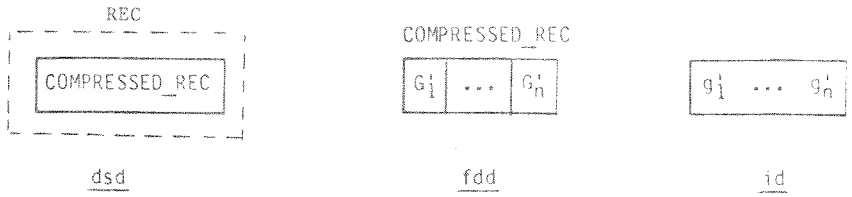


G3. ABSTRACT_REC Type

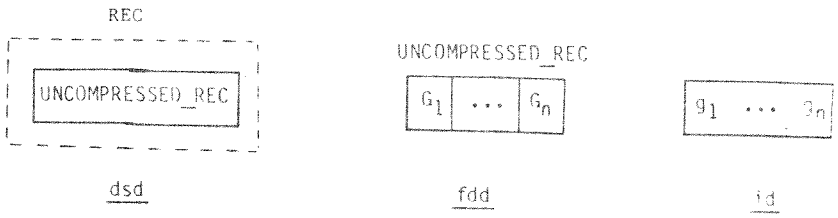


G4. Address Converter Transformation of ABSTRACT_REC Type

The database administrator can declare whether or not instances of REC are to be compressed. (INGRES compresses records by eliminating trailing blanks from character fields). If REC is compressed, it is materialized by the encoding transformation. The COMPRESSED_REC type results (see Fig. G5). If compression does not occur, REC is materialized by the null transformation thereby producing the UNCOMPRESSED_REC type (see Fig. G6). COMPRESSED_REC and UNCOMPRESSED_REC are concrete record types.



G5. Compression of REC Type

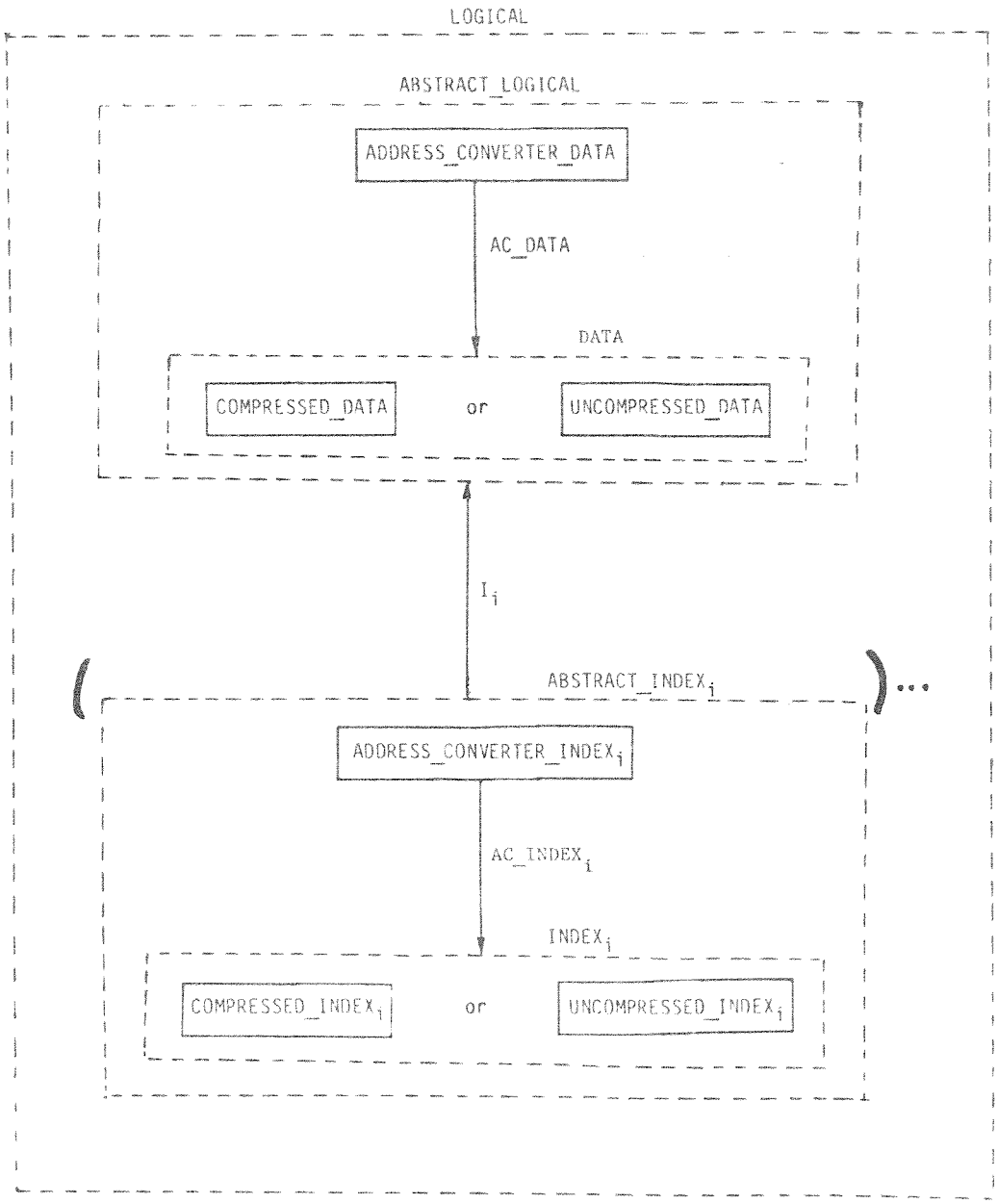


G6. Null Transformation of REC Type

The above sequence of transformations for materializing ABSTRACT_REC will be referred to as the INGRES transformation.

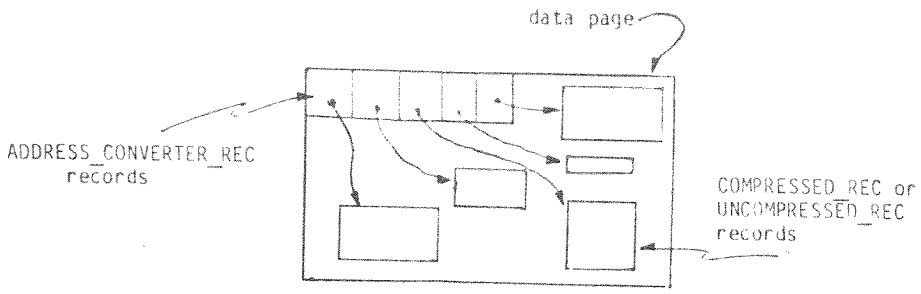
The INGRES transformation materializes both ABSTRACT_INDEX and ABSTRACT_DATA records of Figure G2. The result is shown in Figure G7, along with the rest of the physical architecture of INGRES. As a general rule, INDEX records are not compressed.

The file structures used to organize instances of the concrete record types of INGRES can be understood in terms of the INGRES transformation. When ABSTRACT_REC is materialized, two concrete record types result. One is ADDRESS_CONVERTER_REC. The other is either COMPRESSED_REC or UNCOMPRESSED_REC, depending if records are compressed. In either case, INGRES always stores related records of both types on the same page (see Fig. G8). The storage locations of ADDRESS_CONVERTER_REC records are fixed (but for an exception described below) and do not change with time; the storage location (within a block) of COMPRESSED_REC and UNCOMPRESSED_REC records may change with time. With this notion of parent and child clustering in mind, instances of related record pairs may be organized by indexed aggregate ([Hel76], [Bat82]), hash-based, or unordered file structures. As a general rule, COMPRESSED_INDEX and UNCOMPRESSED_INDEX records are usually organized by indexed aggregate files.[2]



Concrete Record or Linkset	Implementation
ADDRESS_CONVERTER_DATA and COMPRESSED_DATA or UNCOMPRESSED_DATA	hash-based, indexed-aggregate, or unordered
ADDRESS_CONVERTER_INDEX$_i$ and COMPRESSED_INDEX$_i$ or UNCOMPRESSED_INDEX$_i$	hash-based, indexed-aggregate, or unordered
I_i	singular pointer
AC_INDEX$_i$	cellular-sequential singular pointer
AC_DATA	cellular-sequential singular pointer

G7. Physical Architecture of INGRES



G8. Clustering of ADDRESS_CONVERTER_REC and COMPRESSED_REC and UNCOMPRESSED_REC records

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

The physical architecture of INGRES is summarized in Figure G7. Source materials used in the derivation are [Hel75], [Sto76], and [But83].

---

2 The primary keys, or cluster keys ([Bat82]), of indexed-aggregate and hash-based files may be defined over any (nonzero) number of data fields. Whereas indexed compound fields are limited to six or fewer elementary fields, no such size limitation is placed on primary keys.

Appendix III. SYSTEM 2000

SYSTEM 2000 is a product of MRI Systems Corporation. SYSTEM 2000 organizes logical data according to a hierarchical data model. A database is viewed as a collection of disjoint trees that have record occurrences as nodes. Each tree is referred to as a database tree and consists of one root record and all of its dependent records. All database trees are instances of a hierarchical definition tree which specifies the hierarchical relationships among logical record types.[1] 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 shown in Figure S1.

The generic LOGICAL record type supported by SYSTEM 2000 consists of n data fields, $F_1 \ldots F_n$, which are elementary and scalar. Data values, which are stored in these fields, can have variable lengths. Generally, LOGICAL records have variable lengths.

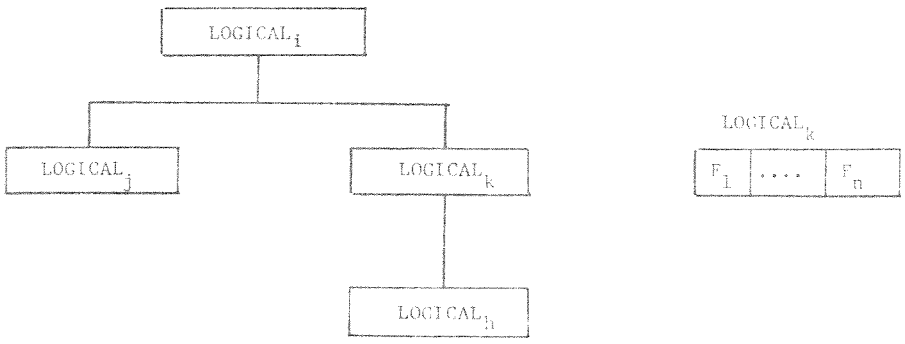


Figure S1. A Representative SYSTEM 2000 Logical dsd.

[1] Terms such as record type, database tree, and hierarchical definition tree are taken from Tsichritzis and Lochovsky ([Tsi77a]). Different releases of SYSTEM 2000 have used different sets of terminology (see [Cas81]).

The first step in the materialization of a hierarchical definition tree is to transform LOGICAL records into ABSTRACT_LOGICAL records by actualizing logical relationships. ABSTRACT_LOGICAL records, it turns out, are fairly easy to understand, but their derivation is long and rather complicated. To make the derivation comprehendible, we will first describe an ABSTRACT_LOGICAL record.

An ABSTRACT_LOGICAL record differs from its LOGICAL record counterpart by the addition of three fields (c.f. Fig. S1.fdd and Fig. S3.fdd). One field, labeled IFN, identifies the LOGICAL record type. A second, labeled P_D, is a parent structure field which contains a pointer to the first child record of a linkset D occurrence. A third, labeled C_A, is a child structure field which contains a parent pointer and a pointer to the next child of a linkset A occurrence. These fields are introduced as a result of the following four step derivation (see Fig. S2).

First, SYSTEM 2000 distinguishes different LOGICAL record types by assigning them distinct internal file numbers (IFNs). Each LOGICAL record is augmented with a field containing its respective IFN.

Second, the relationship between a parent record type and all of its immediate child record types in a hierarchical definition tree is actualized by a single linkset where the roles of parent and child are preserved. The linkset is a multilist with parent pointers. Child records are arranged in reverse chronological order (i.e., new records first, old records last).[2] Figure S2.id illustrates this arrangement. All relationships in a hierarchical definition tree are actualized in this manner.

---

[2] When a SYSTEM 2000 database is first loaded, records are entered in hierarchical sequence ([Cas81]). Subsequent record additions are placed at the head of multilists, so the hierarchical sequencing is not preserved.

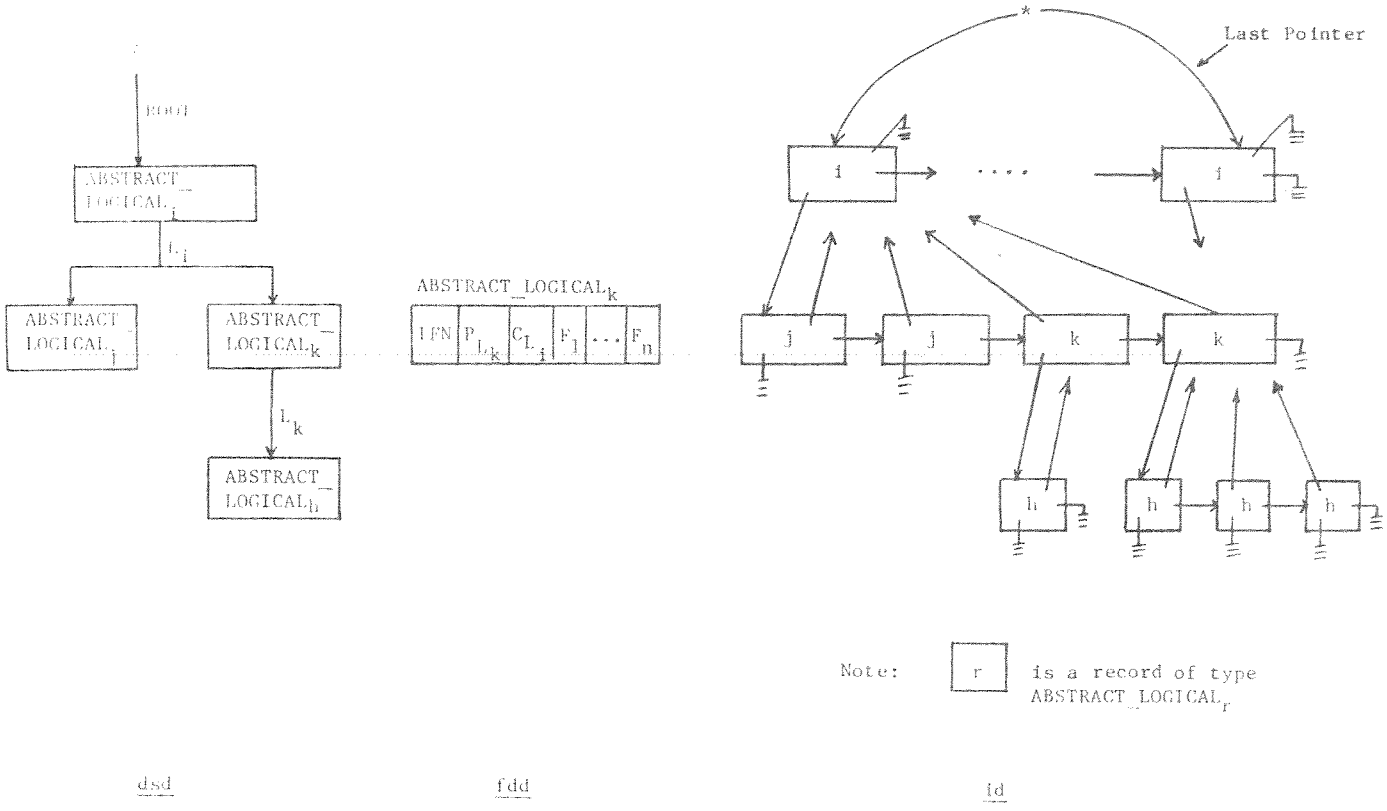


Figure S2. Augmentation, Actualization, and Collection of Logical Records.

Observe that the actualization process introduces a parent structure field in the root record type, a child structure field in leaf record types, and a parent and child field in intermediate record types of a hierarchical definition tree. In order for all ABSTRACT_LOGICAL records to have both parent and child structure fields, some null pointer fields must be introduced. This is done in the remaining two steps.

Third, so that all instances of the root record type can be accessed efficiently, root records are collected together by linkset ROOT. ROOT is a multilist (with precisely one occurrence) with parent pointers and a pointer to the last root record.[3] (Note that a parent pointer to the system * record

[3] SYSTEM 2000 actually stores the pointer to the last root record in the parent pointer slot of the first root record of the ROOT multilist. (Normally, this slot would otherwise be occupied by a null pointer). A more efficient implementation would store the last pointer in the system * record as shown in Figure S2.id.

is indistinguishable from a null pointer). Root records are linked in chronological order (i.e., old records first, new records last).

Fourth, 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 structure field (labeled P_D in Fig. S3.fdd) of a multilist linkset where there are no child records. These null pointers are shown in the occurrence of record types ABSTRACT_LOGICAL$_j$ and ABSTRACT_LOGICAL$_h$ in Figure S2.id.

The generic form of an ABSTRACT_LOGICAL record is shown in Figure S3. An ABSTRACT_LOGICAL record is a child of linkset A (A for ancestor) and is the parent of linkset D (D for descendent). (Note that a specific instance of linkset A is ROOT). A record consists of an IFN field, a parent field P_D, a child field C_A, and n data fields $F_1 \cdots F_n$.

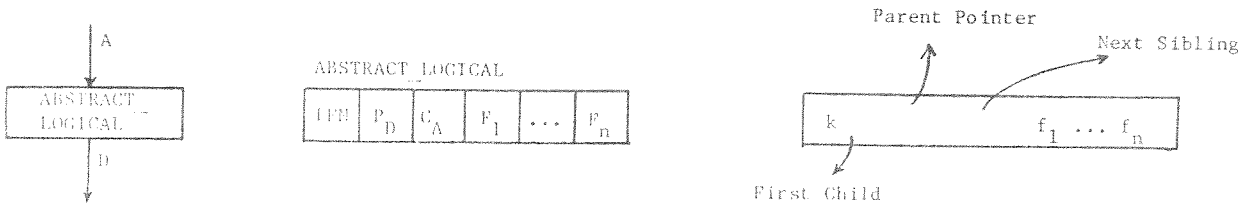


Figure S3. Generic ABSTRACT_LOGICAL Record of SYSTEM 2000.

An ABSTRACT_LOGICAL record is materialized in the following way (see Fig. A4). SYSTEM 2000 indexes all data fields, unless told otherwise. Field F_j is indexed by extracting it from ABSTRACT_LOGICAL records, forming an ABSTRACT_INDEX$_j$ record type. Linkset I_j, which connects ABSTRACT_INDEX$_j$ to ABSTRACT_DATA, is a 1:N inverted list. Pointers of an inverted list are in chronological order. Other fields are indexed in an identical manner.

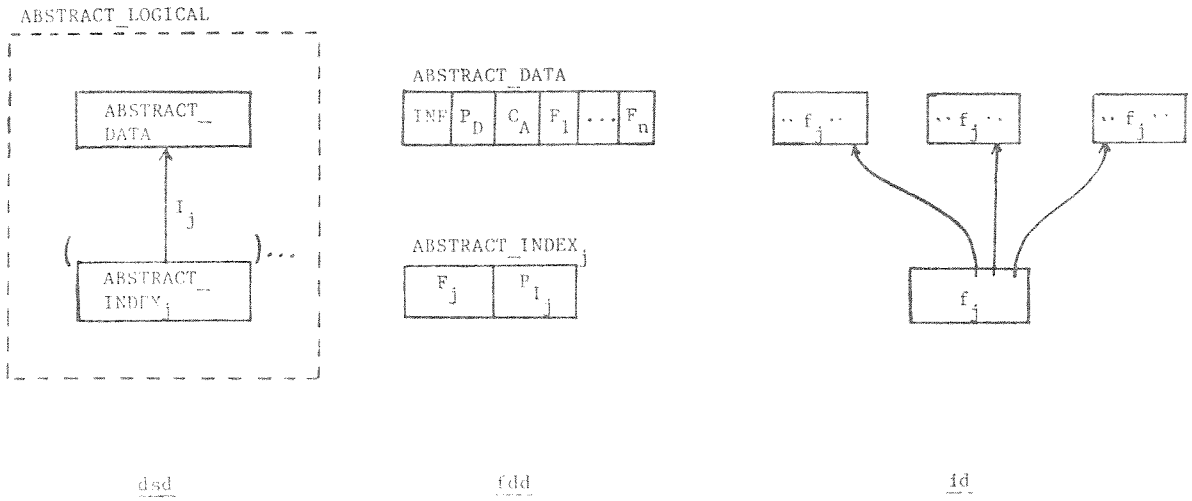


Figure S4. Extraction of Data Fields of ABSTRACT_LOGICAL Records.

Note that if an ABSTRACT_LOGICAL record had no indexed fields, it would be mapped directly to an ABSTRACT_DATA record via the null transform.

ABSTRACT_INDEX$_j$ contains a single data field (F_j) and ABSTRACT_DATA contains n data fields (F_1 ... F_n). SYSTEM 2000 does not store data values directly in these fields, but does the following. When a LOGICAL record type is defined, each data 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 bytes are stored in the field and the remaining bytes are stored as a single EFT_j (Extended Field Table$_j$) record. A pointer connects the "overflowed"

field to the EFT_j record. (Note that a null pointer signifies that the data value is contained entirely within len_j bytes). All data fields are represented in this manner.

This materialization is recognized as a division of a data field F_j into one or two segments. The first segment is stored with the original record; the second, if it exists, is stored as an EFT_j record. Both records are connected by a singular pointer linkset. We will refer to this materialization as the overflow transformation of a data field.

An $ABSTRACT\_INDEX_j$ record is materialized by applying the overflow transformation to field F_j. Linkset V_j is produced in the process. Next, one of two possibilities occur. If the inverted list of field P_{I_j} contains at most one pointer, the $ABSTRACT\_INDEX_j$ record is mapped directly to a DVT_j^* (Distinct Value $Table_j^*$) record via the null transform (Fig. S5).

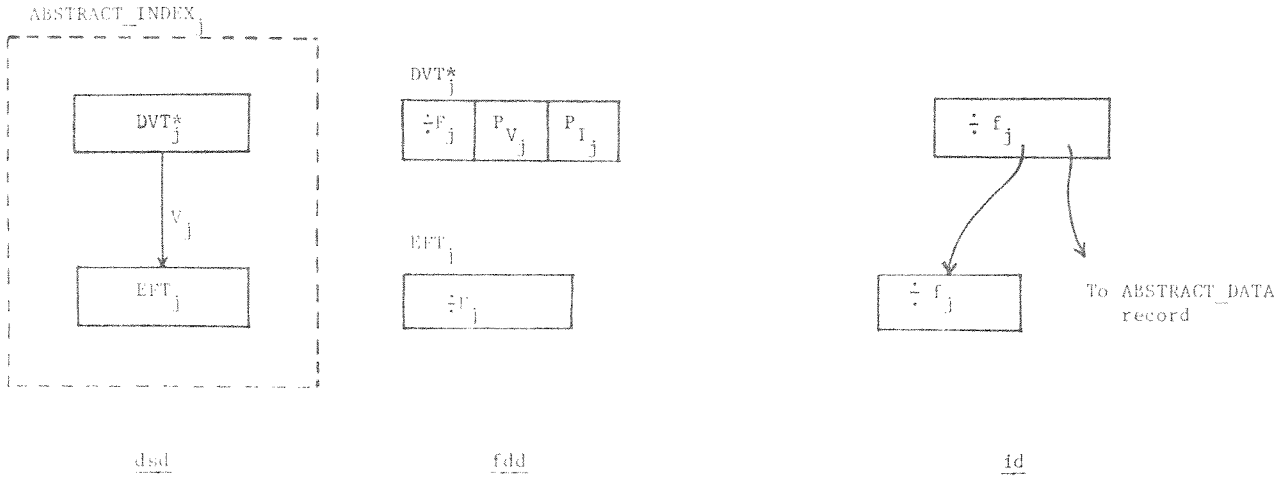
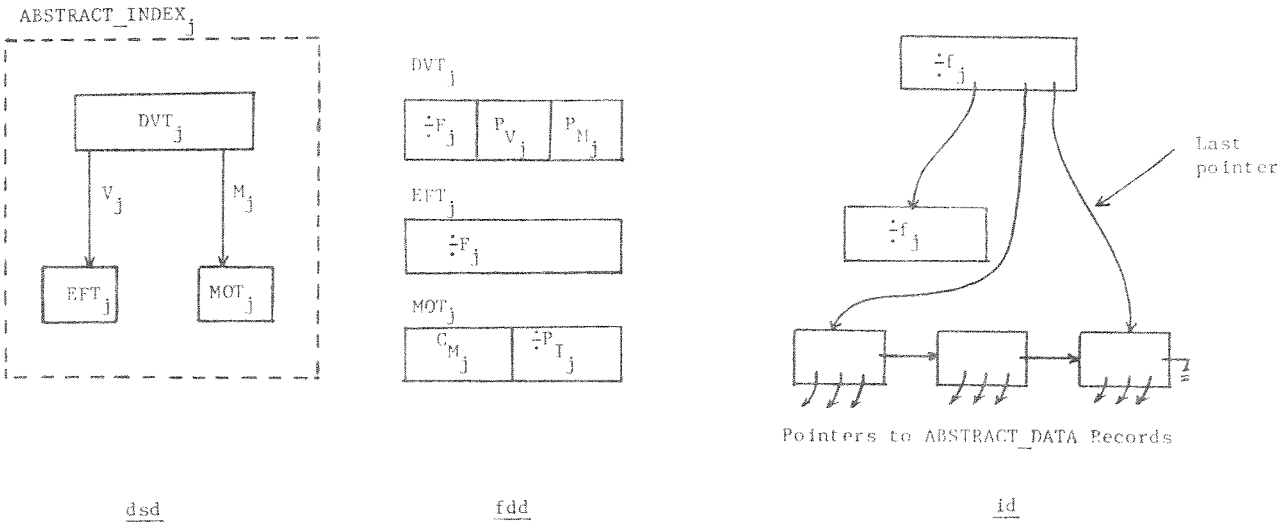


Figure S5. Division of ABSTRACT_INDEX Records.

If the inverted list contains two or more pointers, a DVT_j record is produced by dividing the P_{I_j} field into variable length segments called MOT_j (Multiple $Occurrences\ Table_j$) records. Linkset M_j, which connects DVT_j to MOT_j, is a

multilist with last child pointers.[4] MOT_j records are linked in chronological order (see Fig. S6).[5] Note that the P_{M_j} and P_{I_j} fields in Figure S5 and S6 occur in mutually exclusive situations and that both have the same length. Thus, DVT_j^* and DVT_j records share the same fixed length and format. This is done so that records of both types can be organized by a single file structure.



dsd fdd id

Figure S6. Division of $ABSTRACT\_INDEX_j$

---

4 SYSTEM 2000 actually stores the pointer to the last MOT_j record of an M_j linkset occurrence in the first MOT_j record. A more efficient implementation would store the last pointer in the DVT_j record, as shown in Figure S6.id.

5 MOT records are variable in 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.

An ABSTRACT_DATA record of Figure S4 is materialized by segmenting the IFN, P_D, and C_A fields from the data fields F_1 ... F_n. This produces an HT (Hierarchical Table) record and a DATA record connected by linkset H. H is a singular pointer (see Fig. S7).

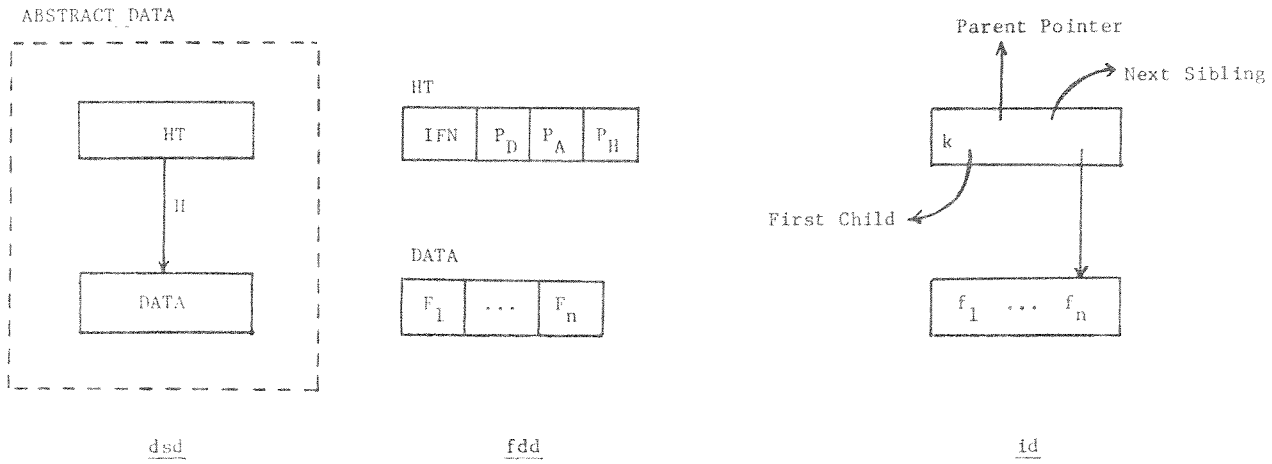


Figure S7. Segmentation of ABSTRACT_DATA Records.

A DATA record is materialized by applying the overflow transform to each of its data fields. The resulting data record is referred to as a DT (Data Table) record; linkset E_j connects it with at most one EFT_j record for each data field F_j. Figure S8.id illustrates a DT record with three data fields that have overflowed. Owing to this general construction, DT records that correspond to a single LOGICAL record type have a fixed length (this is the sum of the n nominal field lengths plus the length of n pointers); EFT_j records have variable lengths. DT records of different LOGICAL record types will, of course, have different lengths.

It is worth noting that if a data value overflows its nominal field length and that it occurs multiple times in a logical database, there will be an EFT_j record for each of its occurrences. No effort is made by SYSTEM 2000 to eliminate duplicate EFT_j records.

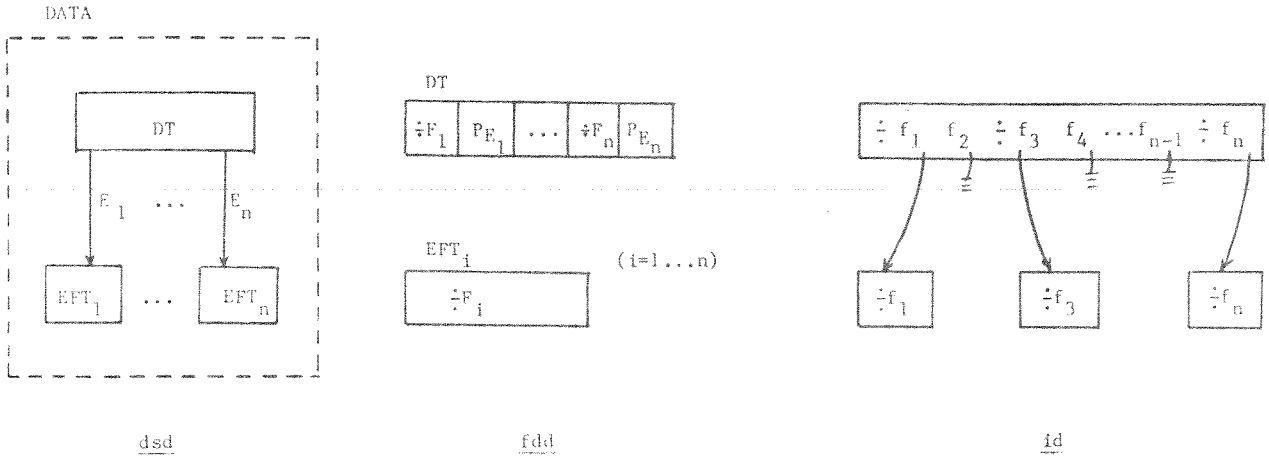
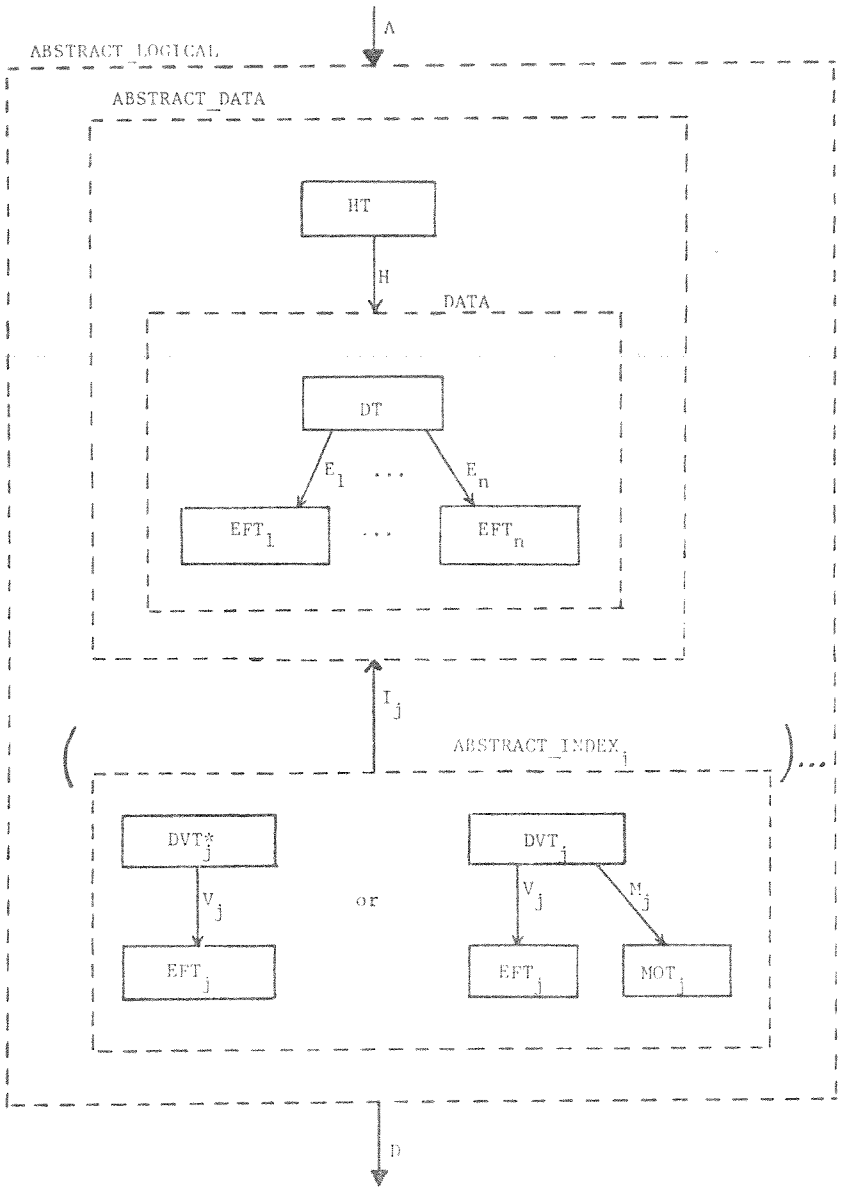


Figure S8. Division of Data Fields of Data Records.

The concrete record types of SYSTEM 2000 are DVT_j^*, DVT_j, MOT_j, HT, DT, and EFT_j. For each j, all occurrences of the DVT_j and DVT_j^* record types are organized by a separate B+ tree. The occurrences of all MOT_j record types are organized by a single unordered file. HT and DT records are stored in distinct unordered files, and the occurrences of all EFT_j record types are stored in a single unordered file.[6]

A data structure diagram that summarizes the derivation of the physical stuctures of SYSTEM 2000 is shown in Figure S9. Source materials used in the derivation are [Cas82], [Cas81], [Tsi77a], and [Kro77].

[6] A SYSTEM 2000 database has one other file, called the DEFIN file. It contains metadata, such as the root nodes of all simple files and the system * record.



dsd

Concrete Record or Linkset	Implementation
HT	unordered
DT	unordered
All EFT_i occurrences for all i	unordered
All DVT_j and DVT^*_j occurrences for each j	B+ tree
All MOT_j occurrences for all j	unordered
H	singular pointer (1:1 pointer array)
E_i and V_j for all i, j	singular pointer
M_j	1:n multilist with last pointer
I_j	1:n inverted list
A, D	1:n multilist with parent pointers (and with last pointer if ABSTRACT_LOGICAL corresponds to root record type)

Figure S9. Abstract and Concrete Record Types of SYSTEM 2000.