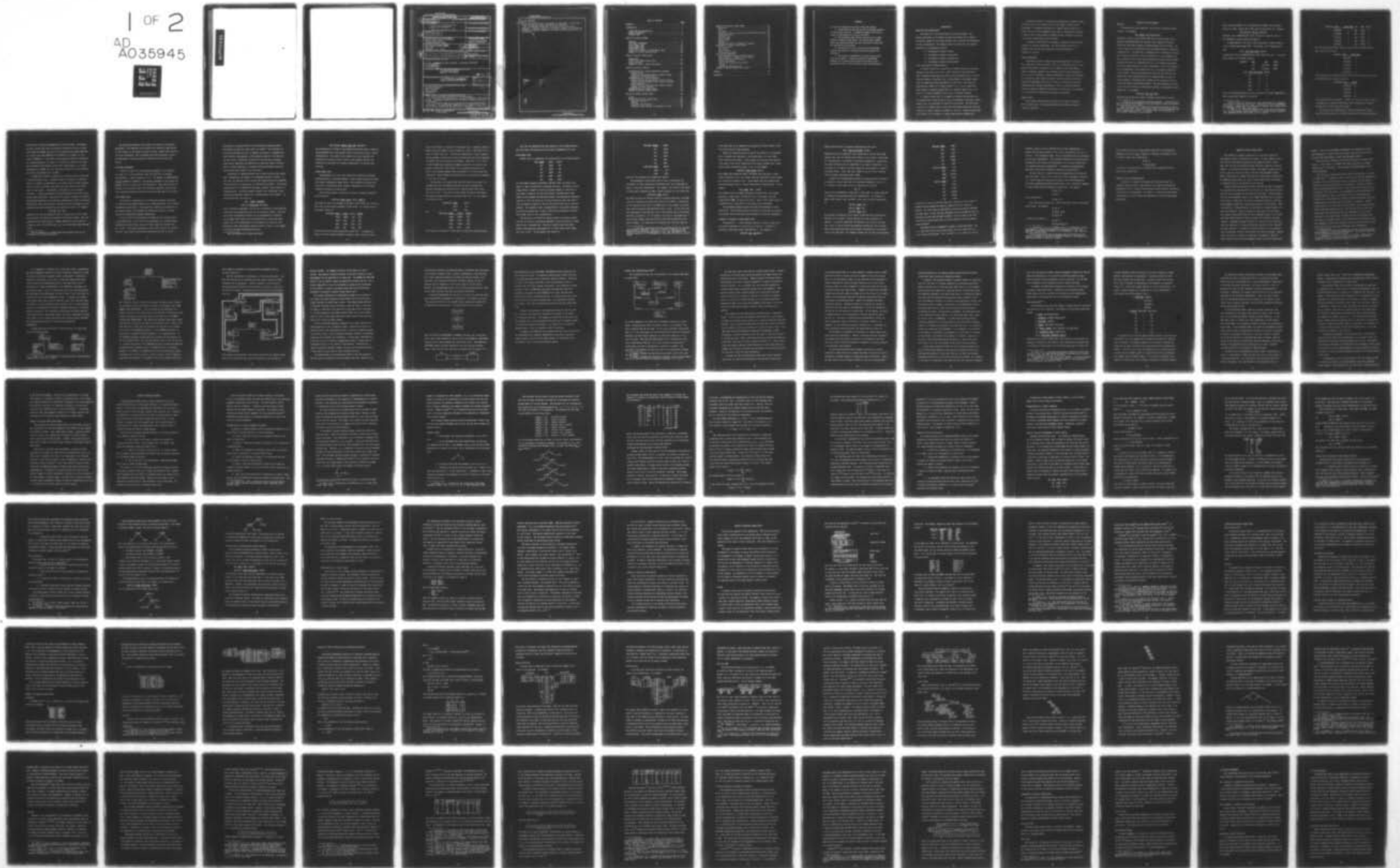
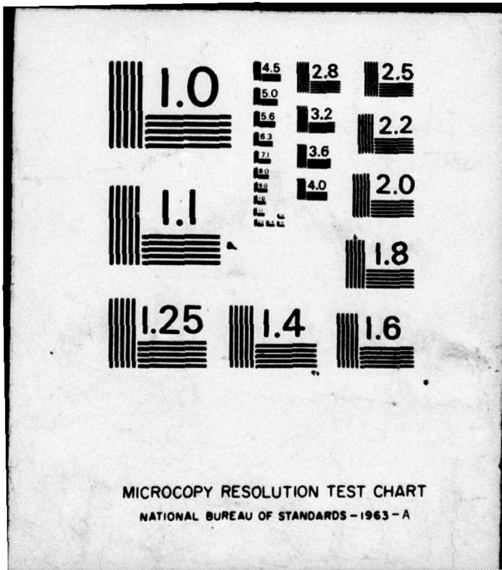
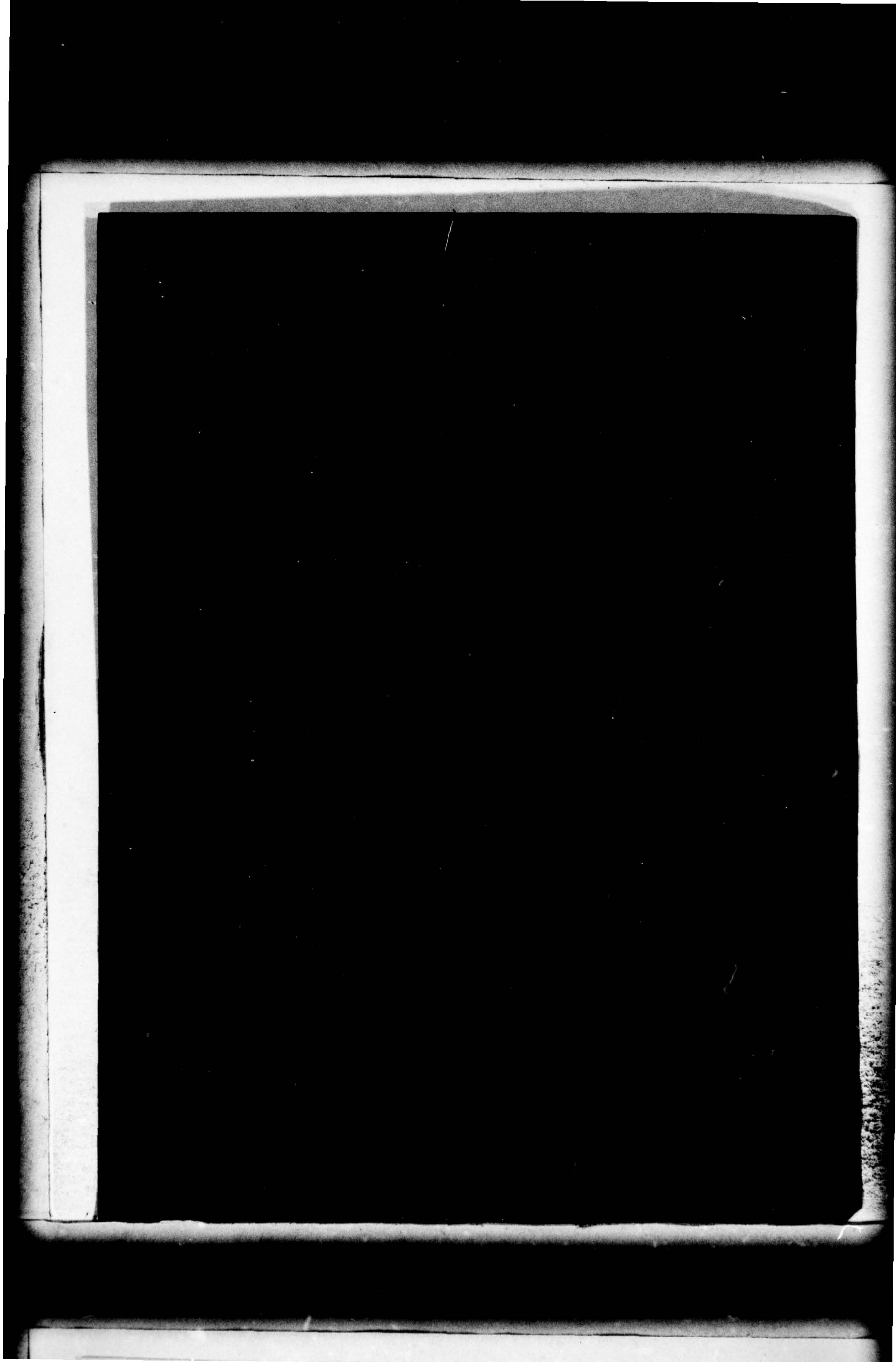

AD-A035 945 DAVID W TAYLOR NAVAL SHIP RESEARCH AND DEVELOPMENT CE--ETC F/G 9/2
DATA BASE DESIGN.(U)
JUN 76 D K JEFFERSON
UNCLASSIFIED DTNSRDC-76-0111
NL
1 OF 2
AD
A035945


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

ADA035945



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) DTNSRDC-76-Ø111	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) DATA BASE DESIGN		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) David K./Jefferson		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS David W. Taylor Naval Ship Research and Development Center Bethesda, Maryland 20084.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element 62760N Task Area TF53531009
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Supply Systems Command Code 0431C Washington, D.C. 20376		12. REPORT DATE (11) June 1976 13. NUMBER OF PAGES 103
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (9) Research and development rept.		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

(12) 1Ø2p.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(16) F53531

18. SUPPLEMENTARY NOTES

(17) TF53531ØØ9

D D C
RECEIVED
FEB 24 1977
A

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data Base Management
Record Structure
File Structure
Optimization

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In the four principal activities of data base design,

(1) Identifiers and descriptors must be grouped together to form logical records. A suggested standard form and procedure for obtaining this form are described.

(2) Relationships among logical records must be established by logical access paths. The hierarchical, CODASYL DBTG, and relational models and the situations in which they are applicable are described.

(Continued on reverse side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

387682

(Block 20 continued)

(3) The physical form of records must be determined. Several techniques for reducing storage requirements are described.

(4) The physical realizations of logical access paths must be determined. A general technique is described, examples are presented and evaluated, and specific suggestions are made for many different types of applications.

ACCESSION for
NTIS

TABLE OF CONTENTS

Page

ABSTRACT 1

INTRODUCTION 2

OBJECTIVES AND ORGANIZATION 2
GENERAL REFERENCES. 3
ABBREVIATIONS 3

DESIGN OF LOGICAL RECORDS. 4

NOTATION. 4
FUNCTIONAL DEPENDENCE 8
FIRST NORMAL FORM 8
SECOND NORMAL FORM. 10
THIRD NORMAL FORM 12
SYNTHESIS OF RECORDS IN THIRD NORMAL FORM 14
SUMMARY OF LOGICAL RECORD DESIGN. 18

DESIGN OF LOGICAL ACCESS PATHS 19

HIERARCHIES 21
CODASYL DBTG OWNER-COUPLED SETS 27
N-ARY RELATIONS 31
SUMMARY OF LOGICAL ACCESS PATH DESIGN 35

DESIGN OF PHYSICAL RECORDS 36

DETERMINATION OF A FIELD'S PRESENCE OR ABSENCE. 37
Huffman Coding 39
Advantages and Disadvantages of Huffman Coding 41
Summary of Field Determination 44
DETERMINATION OF A FIELD'S LOCATION 45
Physical Record Corresponding to Logical Record 45
Logical Record Contained in Many Physical Records. 48
Binary Relations 49
Physical Record Containing Many Logical Records. 51
Summary of Field Location. 52
DETERMINATION OF A FIELD'S VALUE. 52
SUMMARY OF PHYSICAL RECORD DESIGN 55

DESIGN OF PHYSICAL ACCESS PATHS. 56

BLOCKS. 56
CLASSIFYING PHYSICAL ACCESS PATHS 61
Data and Structure 61
Pointers 61
Contiguity and Chaining. 62
Ordering, Direct Access, and Sequential Access 62

EXAMPLES OF PHYSICAL ACCESS PATHS 63
Sequential 63
List 64
Multilist. 64
Analysis of Time for Multilist and Sequential Retrieval. 66
Bounded Multilist. 68
Inverted File. 69
TRIE and TREE. 70
Search Trees 72
Hashing. 77
Generalized Models and Summary of Examples 84
PARAMETERS OF PHYSICAL ACCESS PATHS 87
Block Structure. 87
Free Space 87
Record Design. 88
Hierarchical Storage 88
UTILIZATION PARAMETERS. 89
Rapid Response to Simple On-Line Queries 89
Rapid Response to Complex On-Line Queries. 89
High-Volume Off-Line Retrieval 89
On-Line Maintenance. 90
High-Volume Continuous Maintenance 90
Storage. 91
Loading and Reorganization 91
SUMMARY OF PHYSICAL ACCESS PATH DESIGN. 91

SUMMARY. 92

REFERENCES 93

ABSTRACT

In the four principal activities of data base design,

1) Identifiers and descriptors must be grouped together to form logical records. A suggested standard form and procedure for obtaining this form are described.

2) Relationships among logical records must be established by logical access paths. The hierarchical, CODASYL DBTG, and relational models and the situations in which they are applicable are described.

3) The physical form of records must be determined. Several techniques for reducing storage requirements are described.

4) The physical realizations of logical access paths must be determined. A general technique is described, examples are presented and evaluated, and specific suggestions are made for many different types of applications.

INTRODUCTION

OBJECTIVES AND ORGANIZATION

This report is a very brief survey of data base design. The primary objectives are to present various alternatives in the logical and physical phases of the design process, and to discuss the implications of those alternatives. The design process is divided into four phases, described in corresponding sections:

1) the design of logical records,
2) the design of logical access paths,
3) the design of physical records, and
4) the design of physical access paths.

These terms will be defined briefly.

A "logical record" is a collection of "fields" which is to be manipulated, as an entity, by a user. Each field has, at any particular time, a specific value which may serve to either identify (e.g., a social security number) or describe (e.g., a home address) a real-world object (e.g., an employee) being represented in a data base. The value of a field may be a member of a "simple" domain (i.e., a set of quantities which cannot be logically subdivided) or a "complex" domain (e.g., the value of the field CHILDREN could be a set of children's names).

A "logical access path" is a sequence of logical records which can be followed to get from one object (e.g., an employee) to another, related object (e.g., the department in which he or she works). The term "data structure" is commonly used, but has the unfortunate connotation that the sequence of logical records is inherent in the data. Complex access paths can, however, be the product of complex algorithms and simple data.

A "physical record" is a collection of physically contiguous fields from which some of the fields of one or more logical records can be determined. A frequently used part of a logical record could be on a faster and more costly peripheral device than an infrequently used part. A logical record could contain an employee's age, although the physical record could contain his or her birth date.

A "physical access path" is a sequence of physical records corresponding to a logical access path. The term "storage structure" is commonly used, but has the connotation of being independent of algorithm or process.

GENERAL REFERENCES

James Martin's book, "Computer Data-Base Organization" (Prentice-Hall, Englewood Cliffs, New Jersey, 1975) is an excellent source of both general and specific information on all aspects of data base design. Numerous diagrams, examples, and summaries make the book exceptionally easy to use as a reference; numerous typographical errors will hopefully be corrected in later editions. Donald E. Knuth's series on "The Art of Computer Programming", particularly "Volume 3: Sorting and Searching" (Addison-Wesley, Reading, Massachusetts, 1973) is oriented primarily toward the design of physical access paths; in this area it is unequaled as a collection of techniques and analyses of techniques.

ABBREVIATIONS

Only standard abbreviations are used in this report, except for ACM (the Association for Computing Machinery) and CACM (the Communications of the ACM).

DESIGN OF LOGICAL RECORDS

NOTATION

The notation of n-ary relations will be used to describe logical records.[1] For example,

PART (PART#, PART-DESCRIPTION)

represents a type of record called PART which contains fields called PART# and PART-DESCRIPTION plus possibly other unspecified fields.* Fields must have simple domains; repeating groups are not allowed. One or more fields will be underlined; each continuous underline indicates a "candidate key." By definition a candidate key is a collection of fields which uniquely identifies a record occurrence (i.e., the candidate key has a different value in each record occurrence) and which has no proper subcollection which is a candidate key (i.e., all fields are necessary for unique identification of a record occurrence). One of the candidate keys is designated as the "primary key" (i.e., the primary identifier of the record). Every record must have a value for each field in the primary key; other fields may be null, to indicate that the value is either unknown or irrelevant, but it makes very little sense to have a record describing an unidentified object. The term candidate key refers to the fact that the key, or collection of fields, is a candidate to be a primary key. For example,

EMPLOYEE (ID#, SS#, NAME)

indicates that an employee record is uniquely identified by either a

---

1. Codd, E. F., "Normalized Data Base Structure: A Brief Tutorial," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM, New York, pp. 1-17 (1971).

*The terms "relation," "tuple" and "attribute"are generally used in relational literature rather than "record type," "record occurrence," and "field" respectively, as here. The latter terms are used because of their greater familiarity to most readers.

social security number or by an identification number, but not necessarily by a name. Either ID# or SS# could be the primary key. However,

PART-SUPPLIER (P#, S#, QUANTITY)

indicates that a PART-SUPPLIER record is uniquely identified only by the combination of part number and supplier.

A common domain in two record types establishes a logical association or access path between them.* For example, the following record type

SKILL (ID#, SKILL-CODE, RATING)

is associated with the EMPLOYEE record type through the ID#. If the data base contains the following records:

EMPLOYEE	(ID#,	SS#,	NAME)
	001	123456789	JONES
	002	111223333	SMITH

SKILL (ID#, SKILL-CODE, RATING)

ID#	SKILL-CODE	RATING
001	1	10
001	8	5
002	1	6
002	3	10
002	4	10

then the following information can be obtained by "joining"[2] EMPLOYEE and SKILL records with identical ID# fields:

---

*That is, there is a field in one record type which can be compared with a field in the other record type. Conversion of units is permissible. For example, a weight field in pounds can be compared with a weight field in kilograms, but not with a part number.

2. Codd, E. F., "Relational Completeness of Data Base Sublanguages," Proc. Courant Computer Science Symposium 6: Data Base Systems, Prentice-Hall, Englewood Cliffs, New Jersey, pp. 72-74 (1972).

EMP-SKILL (SS#,	SKILL-CODE,	ID#,	NAME,	RATING)
123456789	1	001	JONES	10
123456789	8	001	JONES	5
111223333	1	002	SMITH	6
111223333	3	002	SMITH	10
111223333	4	002	SMITH	10

and then "projecting" (Codd, 1972, pp. 71-72)[2] onto the SS# and SKILL-CODE fields of the result:

SS#-SKILL (SS#,	SKILL-CODE)
123456789	1
123456789	8
111223333	1
111223333	3
111223333	4

If the records were joined as above and the result projected onto the SS# and RATING fields, the result would be:

SS#-RATING (SS#,	RATING)
123456789	10
123456789	5
111223333	6
111223333	10

The duplicate record is removed. A field, such as ID# in SKILL, which is a candidate key in another record type, such as EMPLOYEE, is called a "foreign key" to that latter record type.

This notation is used because it clearly indicates the logical identifiers and content of the record, and does not constrain the logical

associations or physical implementation of the data base. For example, the SKILL record type could be physically realized as a pair of fields (SKILL-CODE and RATING) that could be a repeating group within EMPLOYEE, or a record type under EMPLOYEE in a hierarchy, or a member of a set owned by EMPLOYEE.* If the SKILL record type were logically represented as a repeating group or as a part of a hierarchy, it would not be logically accessible from another record with a common domain. If it were represented as a member of a set, then there would be no clear indication that ID# is an essential part of SKILL.

A primary key, whether a single field or multiple fields, identifies a real world object (e.g., a physical object or collection of physical objects, a concept, an association, or an event) which is being represented by a record in the data base, while the other fields in the record describe that object. A non-primary, candidate key is an alternate identifier. Primary keys generally behave differently from other fields during update operations: primary keys are added or removed (by insertion and deletion operations on the records), whereas other fields are modified. Other fields, since they represent information about the primary key, are frequently much more volatile than the primary key itself. For example,

PART (P#, COST, QOH)

represents the cost and quantity on hand of a given part, both of which would normally vary much more frequently than the part number. Note that changing the part number may be a complex operation, since P# must be changed not only in the PART record, but in each associated PART-SUPPLIER record.

---

*See the subsection on "CODASYL DBTG Owner-Coupled Sets" for an explanation of the final alternative.

The following subsection will present the concept of "functional dependence." The remainder of the section will consist of applications of this concept to the design of logical records. Simple rules, based on functional dependence, tend to simplify data base description, access, and maintenance, and hence provide an important tool for data base design.

FUNCTIONAL DEPENDENCE

A field A is said to be "functionally dependent" on a collection of fields C if, at any particular time, the value of A is uniquely determined by C. This is written C → A. For example, in SUPPLIER-PART (<u>S#, P#</u>, QUANTITY), QUANTITY is functionally dependent on the primary key <u>S#, P#</u>, but not on either S# or P# individually. (There may be only one supplier for a particular part, but functional dependence concerns only those constraints which must hold for the entire data base at all times.)

FIRST NORMAL FORM

Two sources of complexities in accessing and updating a data base are fields with non-simple domains (e.g., repeating groups), and irregular dependencies among fields. This subsection considers the elimination of non-simple domains; the following two subsections consider a two-step process for eliminating irregular dependencies.

A file is "flat" if all domains are simple and all records consist of the same fields. Such a file can be represented by a two-dimensional tabulation of field values, and is in "first normal form" (Codd, 1971, pp. 11-13).[1] The primary advantages of such a file are that its logical structure can be easily described and understood and that it can be

manipulated by operations which are intuitively and computationally both simple and powerful (Codd, 1972, pp. 65-98).[2] The uniformity of structure encourages the user to organize his queries or programs into simple modules corresponding to the elementary objects in his model of the real world. Furthermore, evolution of the data base is greatly simplified by modularity and simplicity of structure; binary relations may be of great advantage in a rapidly evolving environment since new fields may be easily added to the data base.*

Elimination of repeating groups from a record structure has already been illustrated by the example of the EMPLOYEE and SKILL records. In that case, the employee's SKILL records were associated with his EMPLOYEE record by means of a foreign key, the identification number. The physical representation of the EMPLOYEE record might, however, contain SKILL as a repeating group. As another example, suppose that a department record contains the department number and a repeating group of job numbers. In first normal form this could be:

DEPT (DEPT#, MANAGER#),

DEPT-JOB (DEPT#,JOB#, JOB-DESC).

Given a particular department, its jobs are represented in those DEPT-JOB records with the appropriate value in the foreign key, DEPT#. Note that any n-level hierarchical structure can be represented by n different record types, where level K can be represented by a record type whose primary key is the concatenation of the primary key of level K-1 and additional fields identifying a specific record at level K. For example, to continue the DEPT and DEPT-JOB hierarchy,

*See the subsection on "Binary Relations".

DEPT-JOB-EMP (DEPT#, JOB#, ID#, EMP-DESC).

The proliferation of fields in the primary key does not imply a long and space-consuming key in the physical record, since this is only a logical representation. The fields in the primary key clearly describe the hierarchical structure without arrows or other graphic devices, and therefore are convenient for computer documentation of the data base structure.

SECOND NORMAL FORM

Transforming a file into first normal form eliminates processing complexities due to non-simple domains. Other complexities may be caused by irregular dependencies among fields. This subsection discusses the first step in eliminating those irregular dependencies; the following subsection discusses the second step.

The following example illustrates one type of irregular dependency and its effects:

FURNITURE (NAME, COLOR, PRICE, STOCK #).

Both name and color are required to retrieve a stock number for furniture, but price is functionally dependent on name alone. At a given time the file might consist of:

FURNITURE (NAME, COLOR, PRICE, STOCK#)

NAME	COLOR	PRICE	STOCK#
CHAIR	BROWN	$50	0001
CHAIR	GRAY	$50	0002
DESK	BLACK	$200	0003
DESK	RED	$200	0004

This file has the following undesirable properties: 1) changing the price of an item involves changing a separate logical record for each

color of the item, 2) a new line of furniture (e.g., a cabinet) cannot be added to the file unless a color is known or a null is specified for the color (in which case the record must be replaced when a color is known), and 3) deleting a color will or will not delete name and price, depending on whether that is the only color available. Codd (1971, p.13)[1] refers to these undesirable properties as update, insertion, and deletion anomalies, respectively; their effect is to introduce opportunities for error, since program designers must anticipate all of these exceptional conditions. These problems arise because price is functionally dependent on only part of the key.

The solution is to require that each field which is not part of a candidate key must be determined only by the entire candidate key. Records satisfying this condition are in "second normal form" (Codd, 1971, pp. 13-14).[1] Any record in first normal form may be subdivided into smaller records which are in second normal form. For the example, the result will be:

FURN-PRICE (NAME, PRICE)

NAME	PRICE
CHAIR	$50
DESK	$200

and

FURN-COLOR (NAME, COLOR, STOCK#)

NAME	COLOR	STOCK#
CHAIR	BROWN	0001
CHAIR	GRAY	0002
DESK	BLACK	0003
DESK	RED	0004

The technique is identical to that used in eliminating repeating groups.

Note that the anomalies have been removed; a price change involves only one record, and name and price can exist independently of color.

THIRD NORMAL FORM

Another type of irregularity is illustrated by the following example:

ITEM (<u>ITEM#</u>,	NAME,	PRICE)
001	CHAIR	$50
002	CHAIR	$50
003	CHAIR	$50
004	DESK	$200
005	DESK	$200

The item number uniquely identifies a specific item, but either item number or name is sufficient to determine the price. As before, a price change affects a number of records (an update anomaly), a new type of item cannot be added to the data base unless a (possibly null) item number can be assigned (an insertion anomaly), and deleting an individual item will or will not delete all pricing information about that type of item, according to whether it was the only item of that type (a deletion anomaly). The problem is that, although PRICE is functionally dependent on the entire primary key, it also is functionally dependent on another field, NAME, which is not a candidate key.

The solution is to divide the logical record into smaller records such that no field is functionally dependent on any collection of fields which is not a candidate key. Records in second normal form which satisfy this additional requirement are in "third normal form" (Codd, 1971, pp. 14-15).[1] In the example, the result is:

ITEM-NAME (ITEM#,	NAME)
001	CHAIR
002	CHAIR
003	CHAIR
004	DESK
005	DESK

ITEM-PRICE (NAME,	PRICE)
CHAIR	$50
DESK	$200

Note that the anomalies have indeed been removed.

The advantage of third normal form is that it facilitates the enforcement of those consistency constraints which can be described by means of functional dependencies.[3] For example, the contraint that PRICE is functionally dependent on NAME is easily enforced for the record type

ITEM-PRICE (NAME, PRICE).

Any PRICE field may be modified without violating consistency; each NAME field will continue to be associated with only one PRICE. A new record can be inserted if and only if its NAME field is not already in the data base. This is logically and physically easy to check. On the other hand, the consistency of the ITEM file (not in third normal form) is much more difficult to maintain. If, for example, the price of item number 001, a chair, should change, then the prices of items 002 and 003 would also have to change. If a desk were to be inserted into the file, then its

---

3. It has been argued that functional dependencies do not describe some desirable contraints. Schmid, H. A. and R. J. Swenson, "On the Semantics of the Relational Data Model," Proc. 1975 ACM-SIGMOD International Conference on the Management of Data, ACM, New York, pp. 211-223 (1975).

price would have to be compared with the price of a desk already in the file, which could be time-consuming.

Recall the observation near the end of the subsection on "Notation" that a candidate key identifies a real-world object, and that other fields describe that object. Second normal form requires that descriptions apply to the object identified by the entire candidate key, rather than an object identified by part of it. In the example

FURNITURE (NAME, COLOR, PRICE),

where NAME alone determines PRICE, the PRICE field describes a larger class of furniture (e.g., all chairs) than that identified by the primary key (chairs of a certain color). Third normal form also requires that descriptions apply only to objects identified by candidate keys. In the example,

ITEM (ITEM#, NAME, PRICE)

where NAME (as well as ITEM#) determines the PRICE field, price again describes a larger class of furniture (e.g., all chairs) than that identified by ITEM# (a particular chair). Hence, third normal form is a formalization of a rule which is intuitively quite reasonable.

The preceding subsections have discussed the advantages of third normal form. The following subsection presents a technique for constructing logical record structures from statements of functional dependence.

SYNTHESIS OF RECORDS IN THIRD NORMAL FORM

The notation C A was introduced earlier to indicate that A was functionally dependent on a collection of fields C. Informally, A describes a real-world object identified by C. For example in

EMPLOYEE (ID#, SS#,NAME),

NAME and SS# describe the employee identified by ID#, and in

SKILL (<u>ID#, SKILL-CODE</u>, RATING),

RATING describes the employee's skill identified by ID# and SKILL-CODE. Assume, then, that the desired relationships of the fields in a data base are represented by a collection of functional dependencies. Assume also that each left-hand side of a dependency is as simple as possible - i.e., no fields are included on the left unless they are necessary to identify the proper object. Thus, ID#, SS# —► NAME is true but can be inferred from the simpler expression, ID# —► NAME.

A simple and straightforward method for designing logical records is to associate a record type with each functional dependency; that is, if C —► A then a record type, say R, is constructed as follows:

R(<u>C</u>, A).

This has the disadvantage of requiring a large number of record types and of redundantly representing functional dependencies. For example, if EMP# —► DEPT, EMP# —► LOC, and DEPT —► LOC, then the record types would be

EMP-DEPT (<u>EMP#</u>, DEPT)

EMP-LOC (<u>EMP#</u>, LOC)

DEPT-LOC (<u>DEPT</u>, LOC)

But EMP-DEPT and DEPT-LOC can be joined on DEPT and then projected onto EMP# and LOC to produce a record type with the same information as EMP-LOC. It would be undesirable for EMP-LOC to be a part of the data base, since it can be derived from EMP-DEPT and DEPT-LOC, and the redundancy would complicate update and introduce the possibility of inconsistency in the data base. For example, suppose that the data base contained

EMP-DEPT (EMP#,	DEPT)
001	TOYS
002	TOYS
003	BOOKS
004	BOOKS
005	BABY

DEPT-LOC (DEPT,	LOC)
TOYS	FIRST
BOOKS	SECOND
BABY	FIRST

EMP-LOC (EMP#,	LOC)
001	FIRST
002	FIRST
003	SECOND
004	SECOND
005	FIRST

If EMP-LOC is stored in the data base, then changing the LOC of TOYS to SECOND involves changing one record in DEPT-LOC and two records in EMP-LOC. The latter must be identified by selecting records of EMP-DEPT with DEPT equal to TOYS, and then selecting records from EMP-LOC with the appropriate values (001 and 002) for EMP#. Note that each of the three records is in third normal form; it is the combination that introduces update problems.

The general form of redundancy is simple, as described below. Let E, F, and H be lists of one or more fields, and G be either null or a list of one or more fields. If E →→ F and F, G →→ H, then E, G →→ H is

redundant, since it can be inferred from the other dependencies. A special case is that in which $E \rightarrow F$, $F \rightarrow H$, and hence $E \rightarrow H$ (i.e., G is null). Such redundancies must, then, be eliminated from the collection of functional dependencies. This involves choosing one of the functional dependencies after another and discarding it if it can be derived from the remaining functional dependencies. A reasonably straightforward algorithm is described by Bernstein, Swenson and Tsichritzis.[4]

The next step is to combine functional dependencies with identical left-hand sides. That is, $X \rightarrow Y$ and $X \rightarrow Z$ can be combined to form $X \rightarrow Y,Z$. Record types are constructed as before, with left-hand parts of functional dependencies becoming primary keys. Next, records are combined if their primary keys are dependent on each other. For example,

R1 (<u>X</u>, Y, Z)

R2 (<u>Y</u>, X)

can be simplified to

R3 (<u>X</u>, <u>Y</u>, Z)

This last step may result in a record type which is not in the third normal form.* For example,

R1 (<u>A</u>, E)

R2 (<u>A, B</u>, <u>C, D</u>)

R3 (<u>C, D</u>, E)

is possible and leads to

R1 (<u>A</u>, E)

R4 (<u>A, B</u>, <u>C, D</u>, E)

---

4. Bernstein, P. A., J. R. Swenson, and D. C. Tsichritzis, "A Unified Approach to Functional Dependencies and Relations," Proc. 1975 ACM-SIGMOD International Conference on the Management of Data, ACM, New York, pp. 237-245 (1975).

*This observation was made by Bernstein while delivering the previously cited paper at the 1975 ACM-SIGMOD Conference. The example is a simplified version of Bernstein's example.

and of course R4 is not in third normal form since E is functionally dependent on A only in A, B. However, it suffices to eliminate E from the latter record type, resulting in

R1 (A, E)

R4 (A, B, C, D)

In general, the final step is to eliminate such extraneous fields to ensure third normal form.

SUMMARY OF LOGICAL RECORD DESIGN

In summary, a reasonably straightforward procedure exists for designing logical records. The objective is to begin with functional dependencies, and to combine fields wherever possible without introducing various types of redundancies. The result is a simpler data base whose consistency can be more efficiently maintained by the data base management system.

DESIGN OF LOGICAL ACCESS PATHS

The purpose of a logical access path is to provide a means for identifying a record or collection of records. If every desired collection could be anticipated during data base design and given a unique name, then the user could merely provide the name to specify the data he desired. In practical situations, however, the number of names would be far too large for either the user or the data base management system. Hence, access paths to complex collections of records must be constructed by combining simple paths; the forms of these simple paths and the ways of combining them are the subjects of this section.

Logical access paths which are short and match the user's conception of his problem are very desirable for the user, in order that the data he requires can be meaningfully and succinctly described. The efficiency of the data base management system is also highly dependent upon the logical access paths. Paths which are long and presented piecemeal to the data base management system will require many physical record accesses. Short access paths, or long paths which are described in a single request, are much more desirable. Even if realized by long physical access paths, such logical access paths allow the data management system to read ahead and retain needed records in faster storage, thus improving real-time response. Seemingly unrelated requests for records are most inefficient, while the retrieval of a large well-defined collection can be optimized and quite efficient. In general, the more difficult it is to express a user's intentions, the more inefficiently they will be realized. However, there is a price to be paid for simplicity in expression: physical access paths and the data base management system itself may become quite

complex. This is not surprising, considering the complexities of data base management; if less is done by the user, then more must be done by the data base administrator or data base management system.

This section will describe three different models of logical access paths: 1) hierarchical, 2) CODASYL DBTG owner-coupled set (or simple network), and 3) relational. The advantages and disadvantages of each will be discussed.

This section is concerned only with logical descriptions - or views - of a data base.[5,6] Different views of the same data base are possible and desirable, since they permit each user to be unaware of those records, fields, and associations which do not concern him. Security is facilitated by providing different views - a user cannot access data not in his description of the data base. Data independence is also facilitated by providing different views - the program's view of the data base can frequently be held constant even though the underlying physical data base changes because of new applications, hardware, or software. Finally, on-line access to the data base by non-programmers may be greatly facilitated by providing different views - each user can view the data base through names and associations which are meaningful to him, but not necessarily to the other users. However, the data base administrator must define the different views. One consideration in evaluating a data base management system, then, should be the ease with which views can be defined and the efficiency with which they can be implemented.

---

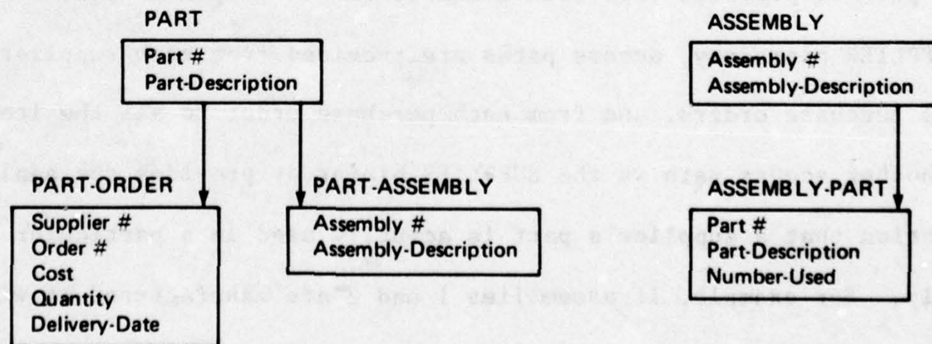
5. Boyce, R. F., and D. D. Chamberlin, "Using a Structured English Query Language as a Data Definition Facility," IBM Research Laboratory, San Jose, California (December 10, 1973).

6. Chamberlin, D. D., J. N. Gray, A. L. Traiger, "Views, Authorization and Locking in a Relational Data Base Design," IBM Research Laboratory, San Jose, California (October 7, 1974).

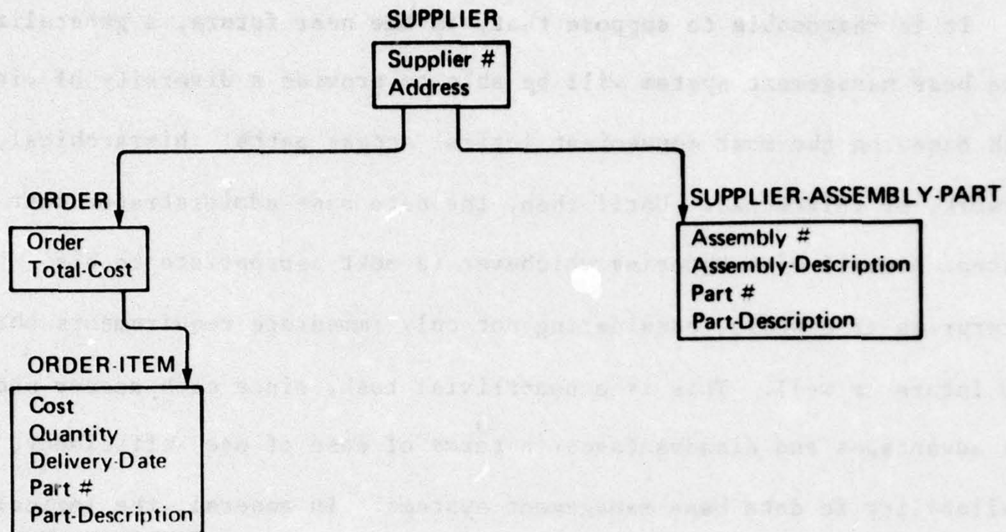
It is reasonable to suppose that, in the near future, a generalized data base management system will be able to provide a diversity of views, each based on the most convenient logical access paths: hierarchical, network, or relational. Until then, the data base administrator must content himself with choosing whichever is most appropriate to his enterprise as a whole, considering not only immediate requirements but the future as well. This is a non-trivial task, since each access model has advantages and disadvantages in terms of ease of use, efficiency, and availability in data base management systems. In general, the logical access model which least restricts physical access, the relational model, is easiest to use and potentially most efficient, but is least available because of the complexities involved in mapping from logical to efficient physical structures. The hierarchical access model introduces physical access paths early in the design process, but has the advantages of being very simple to use in simple applications and of being readily available. The following subsections will clarify these generalities.

HIERARCHIES[7]

A sample data base consisting of three hierarchies is shown below and on the following page:



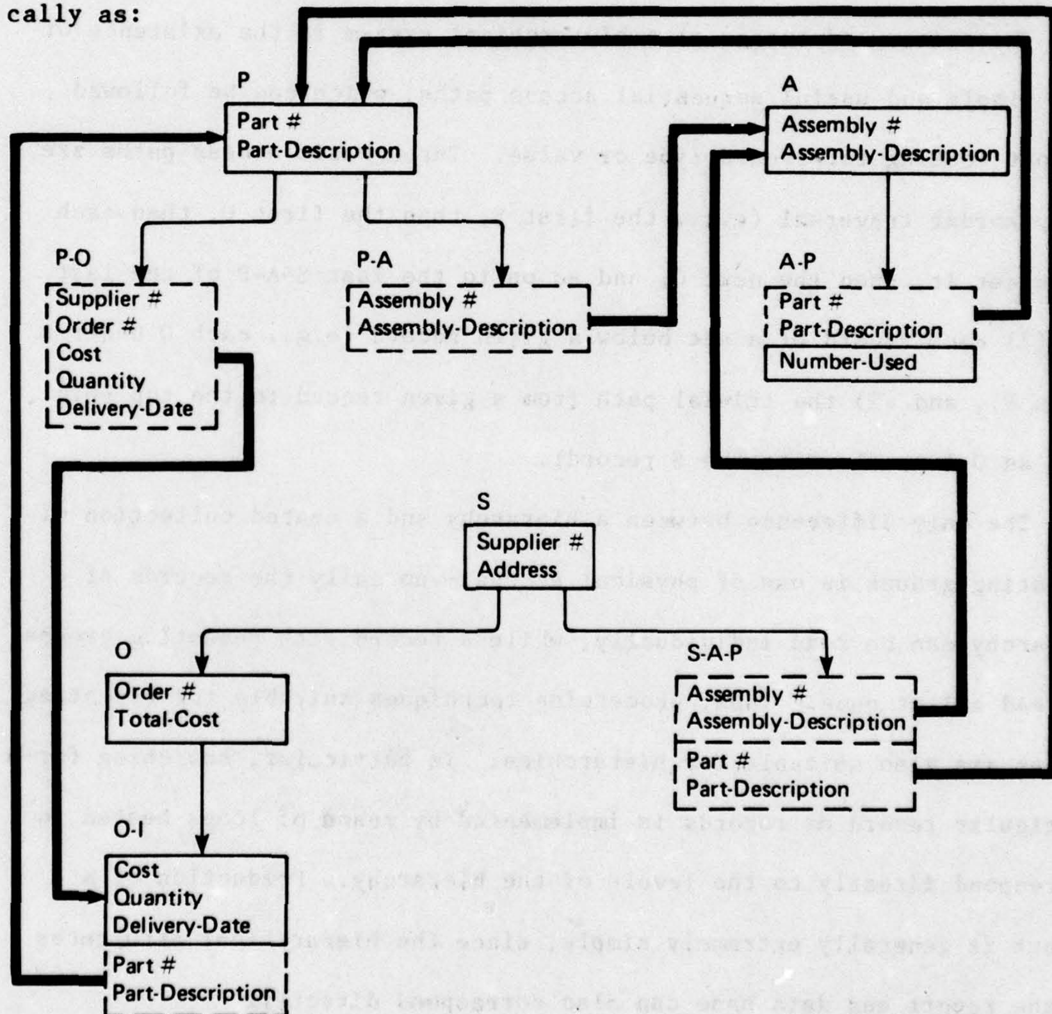
7. Date, C. J., "An Introduction to Database Systems", Addison-Wesley, Reading, Mass. (1975), pp. 137-221.



The rectangles indicate logical record types; the name of each is above the rectangle, and the field names are within the rectangle (e.g., PART (PART#, PART DESCRIPTION)). An arrow indicates the association of a single occurrence of the record type at the tail with a collection of zero or more occurrences of the record type at the point (e.g., one ORDER contains many ORDER-ITEMs). The name of the top-most record type in each hierarchy will be used as the name of the hierarchy itself. Thus, in the PART hierarchy, access paths are provided from each part to all orders for it and to all assemblies using it. In the ASSEMBLY hierarchy, an access path is provided from each assembly to its component parts. In the SUPPLIER hierarchy, access paths are provided from each supplier to all his purchase orders, and from each purchase order to all the items on it. Another access path in the SUPPLIER hierarchy provides the additional information that a supplier's part is actually used in a particular assembly. For example, if assemblies 1 and 2 are manufactured at widely separated sites but both use part 3, then a supplier of part 3 might be associated with assembly 1 or assembly 2, but not necessarily with both.

(This might be convenient in tracing defective assemblies back to the part suppliers.)

Note the duplication of information in the three hierarchies - the data base description is concerned only with logical structure, where redundancy may be acceptable. Physical redundancy is controlled by putting duplicated fields into one hierarchy and pointers to them into the other hierarchies. The previous data base could be realized physically as:



Record names are abbreviated. The dotted records have the logical fields shown but are physically only pointers (indicated by dark arrows) to

physical records. For example, the S-A-P record points to A and P records. The physical records are placed in whichever hierarchy is most advantageous for the operation of the data base. The ORDER# and SUPPLIER# fields of the P-O logical record are physically obtained from the O and S records respectively, which are accessible by going up the S hierarchy. In this example, a report on purchase orders will be more efficiently processed if arranged by supplier rather than by part.

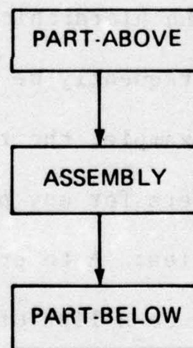
The primary advantage of a hierarchical system is the existence of very simple and useful sequential access paths, which can be followed without testing for record type or value. Three basic access paths are (1) preorder traversal (e.g., the first S, then the first O, then each O-I under it, then the next O, and so on to the last S-A-P of the last S), (2) each record of a set below a given record (e.g., each O under a given S), and (3) the trivial path from a given record to the top (e.g., from an O-I to the O to the S record).

The only difference between a hierarchy and a nested collection of repeating groups is one of physical access - normally the records of a hierarchy can be read individually, while a record with repeating groups is read all at once. Thus, processing techniques suitable for repeating groups are also suitable for hierarchies. In particular, searching for a particular record or records is implemented by means of loops nested to correspond directly to the levels of the hierarchy. Production of a report is generally extremely simple, since the hierarchical structures of the report and data base can also correspond directly.

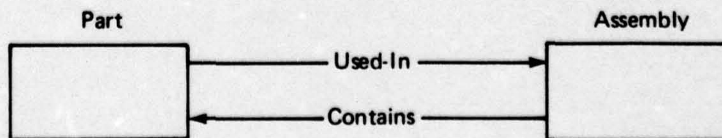
One disadvantage of the hierarchical model is that the process of physical data base design may become extremely difficult, because of the

complications involved in eliminating physical redundancy while providing for a variety of logical views. Another disadvantage is that there may be no clear separation between the logical and physical design of the data base - the reduction of physical redundancy through the use of pointers must be foreseen at a very early stage of logical design. However, both disadvantages affect just the data base administrator; they do not affect the user, who sees only logical hierarchies.

A possible disadvantage for the user is that not all real-world data can be conveniently represented in hierarchies. For example, the query "list each part which is in an assembly which contains part number 001" is inherently non-hierarchical. A hierarchical structure would be:



where the arrow from PART-ABOVE to ASSEMBLY indicates that a given part may be used in many assemblies, and the arrow from ASSEMBLY to PART-BELOW indicates that a given assembly may contain many parts. PART-ABOVE and PART-BELOW contain the same information, but must be given different names to make the structure look like a hierarchy when in reality it is



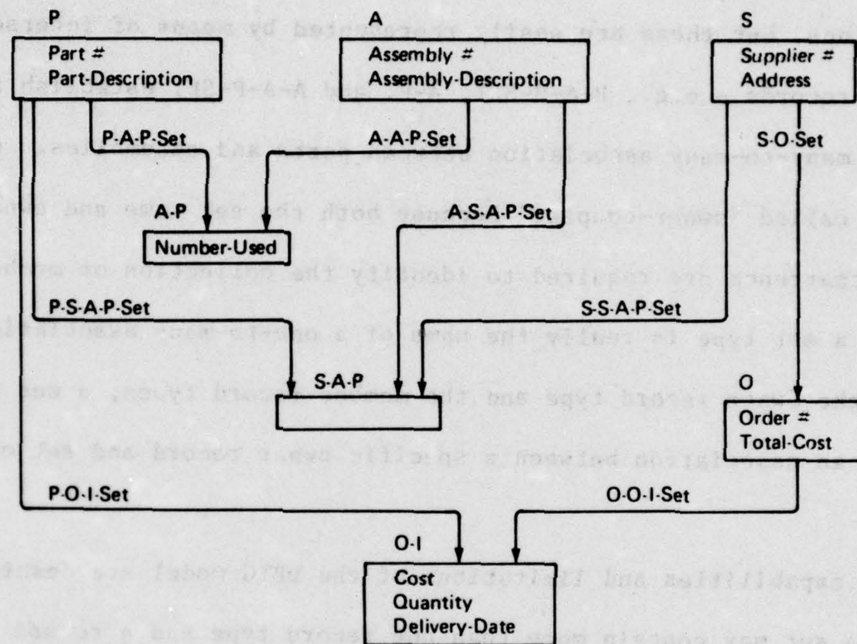
Such distortions as the PART-ABOVE, PART-BELOW artifice complicate the user's view of the data. Furthermore, adding another logical hierarchy to the data base would involve additional physical pointers. Resolving the query in the sample data base is also unsatisfactory, since it involves both the P and A logical hierarchies: the P hierarchy is used to obtain a set of ASSEMBLY#s associated with PART# 001, then the A hierarchy is searched for each ASSEMBLY# in the set, and then each associated P record is obtained directly. No logical access path can be used to satisfy this query, although there is an appropriate physical access path. Note that the searching involved may be quite time-consuming.

A further disadvantage of the hierarchical model is that the need for a logical access path must frequently be foreseen in the design of the logical data base. For example, the resolution of the query "list each supplier who is filling orders for any part used in assembly number 1" involves two logical hierarchies: A to produce a set of parts in assembly number 1, and P to produce a list of suppliers.

In summary, the simplicity of a hierarchichal data base is attractive, but considerations of physical implementation may greatly complicate logical design, and even fairly simple queries, if unforeseen or non-hierarchical, may be costly and logically complex.

CODASYL DBTG OWNER-COUPLED SETS[8,9]

The preceding data base can be represented in the CODASYL DBTG model as shown below:



As in the diagrams of the hierarchies, rectangles correspond to record types. An arrow points from a "set owner" type to a "set member" type, and is labelled with the set name. An A-P record represents intersection data - i.e., that the owners of the P-A-P-SET and A-A-P-SET are associated. Similarly, an S-A-P record represents an association among an S, an A, and a P record. Note that the DBTG diagram clearly expresses the structural relationships involved: P, A, and S records are associated in various ways, but none is subordinate or superior to another. The equivalent relationships are much less clear in the hierarchical diagram.

---

8. CODASYL, "CODASYL Data Base Task Group Report, April 1971", ACM, New York (1971).
9. CODASYL, "CODASYL Data Description Language, Journal of Development, June 1973," National Bureau of Standards, Washington, D.C. (1973).

The DBTG model may be described as a simple network model - network because any two record types can be associated, and simple because the association must be one-to-many. Complex networks allow many-to-many associations, but these are easily represented by means of intersection sets and records - e.g., P-A-P-SET, A-P, and A-A-P-SET establish an indirect many-to-many association between parts and assemblies. DBTG sets are called "owner-coupled" because both the set name and owner record occurrence are required to identify the collection of members. That is, a set type is really the name of a one-to-many association between the owner record type and the member record types; a set occurrence is an association between a specific owner record and set of members.

The capabilities and limitations of the DBTG model are described below. A set may contain more than one record type and a record may be the owner of more that one set type (e.g., P owns P-O-I-SET and P-A-P-SET). However, a set occurrence can have only one owner and a record occurrence can be a member of only one occurrence of a given set type - hence, a set name alone uniquely determines an access path from member to owner. Furthermore, a record occurrence can be the owner of only one occurrence of a given set type - hence, a set name alone also uniquely determines an access path from owner to a set of records related to it. These restrictions also simplify physical record structure; a record type could, for example, be allocated one pointer for each owned set type and one pointer for the owner of each set type containing it.

Although the DBTG and hierarchical data bases contain identical data, there are important differences in the available access paths.

A hierarchical data base is, in some respects, a special case of a DBTG data base in which a record type can be a member of only one set type. Hence, in the hierarchical model, the set name need not be specified to uniquely determine an owner occurrence (there can be only one record above any other) or a set of members of a given type (there can be only one such set below any owner record). Compare the hierarchical and DBTG diagrams and note that names are needed only on the arrows of the latter. However, the hierarchical data base has implied access paths which are not present in the DBTG data base: namely, the preorder traversals referred to earlier. Sequential access to a collection of records is defined for only two cases in the DBTG model: for all the records of a set and for all the records of a record type. In both models, the order of records within a set is determined at the time records are inserted into it. The place of insertion may be determined by content, by time of insertion (last-in, first-out, or first-in, first-out, for example), or by an arbitrary procedural decision (i.e., a procedure can find a particular record occurrence and insert before or after it).

The locations of rectangles are immaterial in a DBTG diagram (only named arrows are significant). However, locations determine the logical and possibly even the physical structure of a hierarchical data base, since the relative locations determine the sequence of records processed in the preorder tree traversal.

The Sub-Schema Data Definition Language can be used to define hierarchical substructures of a network data base, but there are no Data Manipulation Language commands to manipulate entire hierarchies. Some of the same effects may be obtained, however, by the use of automatically

invoked procedures to, for example, delete everything below selected records when those records are themselves deleted.

In general, the increased freedom of movement allowed in a DBTG data base is at the cost of increased complexity in processing hierarchies. However, queries such as "list each part number which is in an assembly which contains part number 001," mentioned earlier, are much more easily and naturally expressed in the DBTG system than in a hierarchical system. The query could be satisfied by following an owner to member to owner path from PART# 001 through A-P records to each associated A record, then following an owner to member to owner path to each associated P record. The entire path has four links, two one-to-one and two one-to-many (requiring two loops). No searching is necessary. The equivalent query in the hierarchical model involved two one-to-many links, plus a search for each of a set of ASSEMBLY#s. The other query presented earlier, "list each supplier who is filling orders for any part used in assembly number 1," is somewhat complicated in the DBTG data base - its resolution requires an access path from ASSEMBLY# 1 to the members of an A-A-P-SET, to the owner of a P-A-P-SET, to the members of a P-O-I-SET, to the owner of an O-O-I-SET, and finally to the owner of an S-O-SET. Again, as in the case of the hierarchical model, the need for a particular access path, from A to S, was not foreseen in the logical data base design.

A particularly important difference for the present discussion is that the DBTG system appears to more effectively separate logical and physical descriptions of a data base, while at the same time allowing reasonable flexibility in the choice of physical structures. For example, if the user of a hierarchical data base must deal with logically redundant

data, then performing an update requires knowledge of whether the data is physically redundant or not; he should not perform updates of many logical records if only one physical record is involved. For the DBTG user, redundancy can be eliminated from the logical data base.

In summary, the DBTG model provides more flexibility in the design of logical access paths than is provided by the hierarchical model. Simple paths, such as a preorder tree traversal, may be somewhat more complicated in the DBTG model than in the hierarchical model.

N-ARY RELATIONS[10,11]

The preceding section entitled "Design of Logical Records" introduced the notation of n-ary relations. The example of the previous subsections becomes

P (PART#, PART-DESCRIPTION)

A (ASSEMBLY#, ASSEMBLY-DESCRIPTION)

S (SUPPLIER#, ADDRESS)

O (ORDER#, SUPPLIER#, TOTAL-COST)

O-I (PART#, ORDER#, COST, QUANTITY, DELIVERY-DATE)

A-P (ASSEMBLY#, PART#, NUMBER-USED)

S-A-P (SUPPLIER#, ASSEMBLY#, PART #)

Records must be in first normal form. The records in the data base are unordered, although an order based on record content may be imposed upon a collection of records when it is retrieved (there will, of course, be

---

10. Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM, New York, pp. 35-68 (1971).

11. Chamberlin, D. D. and R. F. Boyce, "SEQUEL: A Structured English Query Lanuage," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM, New York (1974).

an order imposed by physical structure, but this is subject to change whenever the data base is reorganized). Duplicate records are automatically removed from the data base. A logical access path from one record type to another is established by the existence of comparable fields (i.e., fields with common domains) in the two record types. For example, supplier number is a foreign key in O which establishes a one-to-many association between S and O:

S (SUPPLIER#, ADDRESS)

1	ARLINGTON
2	ALEXANDRIA
3	BETHESDA

O (ORDER#, SUPPLIER#, TOTAL-COST)

101	1	100
102	1	20
103	1	500
104	2	50
105	3	80
106	3	100

The resolution of the first query discussed earlier, "list each part number which is in an assembly which contains part number 001" requires only A-P records. The assembly numbers of all records containing part number 001 are used to identify A-P records which have the desired part numbers. The other query, "list each supplier who is filling orders for any part used in assembly number 1," requires A-P records to determine part numbers used in assembly number 1, O-I records to determine order numbers for those parts, and O records to determine the supplier numbers.

The subsection entitled "Synthesis of Records in Third Normal Form" discussed the rules for designing records in a relational data base. These rules allow the designer only a very limited degree of flexibility in the choice of record structures: all that he can do, in fact, is determine the order in which redundant functional dependencies are eliminated. Otherwise, the logical record structures will be completely determined by the functional dependencies. Since all logical access paths are determined implicitly by the contents of the logical records rather than by explicit structures, the functional dependencies determine logical access paths as well.

In the hierarchical model, there is a natural access path (with order determined by the logical data structure, content, time of insertion, or an arbitrary procedure) through an entire hierarchy. In the DBTG model, there is a path (with order determined by record content, time of insertion, or an arbitrary procedure) through a set occurrence or record type. In the relational model, all paths are determined by record content alone; order is determined (by content) at the time a collection of records is retrieved, rather than at the time a record is inserted. All record occurrences in a relational data base are unique; identical content of two record occurrences must indicate redundancy, since the two cannot be distinguished by set associations or by order. Identical records can exist in a hierarchical or DBTG data base.

Information can be represented by record content, associations, or order in a hierarchical or DBTG data base, but only by record content in a relational data base. The discipline of the relational model has a definite advantage: data base definition and manipulation can be a

science, rather than an art. There is an accompanying disadvantage, however: hierarchies and networks often correspond more closely than do relations to our conceptions of the real world.

The relational model has the further disadvantage that modifying a candidate key in a record can have far-reaching consequences, since all records associated with it by means of a foreign key will also have to be updated. This follows from the interpretation of a candidate key as the identifier of a real-world object; if the name of an object is changed, then all information concerning it must refer to the new name. Hence, it is desirable to require that object identifiers be inserted or deleted, but not changed. This may not accurately model the real world, where identifiers do change (women may change their names through marriage, part numbers are changed, and so on). In the hierarchical and network models, common data may be factored out of a collection of records and stored in a higher hierarchical level; in particular, the identifier of an object can be separated from the records describing it, so it need logically be changed in only one place.

A significant advantage of the relational model is that logical access paths are effectively unconstrained - at run-time the user can establish any association logically implied by the data. Furthermore, the operations available in relational data base management systems allow the expression of quite complex requests in single statements, allowing the system to perform various optimizations based on the existing physical access paths.

A disadvantage of the relational model is that the freedom allowed in constructing logical access paths demands a data base management system capable of performing much of the optimization normally performed

by the data base designer. The user is not constrained to do things "efficiently," for example, by means of pre-defined sets, so the system must somehow translate his "inefficient" statements into a form compatible with physical structures (or transform the physical structures into structures more compatible with the user's statements). Accordingly, relational data base management systems are at present generally less available and less efficient than hierarchical or DBTG sytems.

SUMMARY OF LOGICAL ACCESS PATH DESIGN

Many applications involve queries which are sufficiently ad hoc to defy optimization by data base designers; such queries are best handled by relational systems. Other applications, althoug.. relatively predictable and simple, still demand high efficiency; for these a hierarchical system would be most suitable. Finally, other applications are predictable, complex, and require high efficiency; these would require a DBTG system.

What, then, of the many real environments, in which all these applications are mixed with others which are unsuitable for any of the three types of systems? The thesis to be developed in the following sections is that data base design is best accomplished if decisions are postponed as long as possible - hence, that system is best which most effectively accommodates changes in logical requirements and physical implementation. At present, the relational model most easily accommodates changes in logical requirements, and hence is to be preferred in the early stages of logical design. The DBTG model is widely available and most easily accommodates changes in physical implementation, so it is to be preferred in the final stages of physical design.

DESIGN OF PHYSICAL RECORDS

An access path provides a way to locate a desired record; this section discusses what a record looks like when it is located and read or written. In other words, it describes how information can be stored to reduce the costs of mass storage, computer time, and user time. Generally, the emphasis of this section will be on ways of reducing physical redundancy and thereby reducing mass storage and updating costs, at some increase in computer time and response time. The following section on access paths will consider techniques which reduce response time but generally increase the amount of mass storage.

This section will present possible answers to three basic questions in the design of physical records:

1. What schemes can be used to indicate the presence or absence of optional fields? (In an address file, for example, some people have apartment numbers and some do not.)

2. Where is the value of a given field? (To reduce transfer time, frequently and infrequently used fields from one logical record may be in different physical records.)

3. How is the value to be interpreted? (Compression schemes may be used to reduce storage space.)

The intent is to list the techniques applicable to physical record design and to organize them into a readily comprehensible form. However, these techniques are most useful if considered within the context of the total problem of data base design. Physical record design is not a completely isolated activity; some techniques of data compression, for example, imply restrictions on physical access paths.

The term "control field" will be used to denote a field which is part of a physical record, but which is accessed only by the data base management system and not by the user. The term "physical record" implies physical contiguity. A logical record may consist of many physical records linked together by pointers. For example, binary relations have been used to implement an n-ary relational system.[12] Furthermore, pointers need not be physical addresses, but may be key values or relative addresses.

DETERMINATION OF A FIELD'S PRESENCE OR ABSENCE

At least six possible methods exist for determining whether or not a given field is actually present in a record occurrence.

- First, a field may be known to be logically present at all times (a fixed field).
- Second, a field may be known to be present at all times within a repeating group.
- Third, the combination of optional fields which are actually present may be indicated by an encoded control field.
- Fourth, the presence or absence of an individual field may be indicated by a bit in a control field.
- Fifth, a field may be physically present at all times but considered logically absent if it has a particular (null) value (e.g., an illegal character).
- Sixth, each field present may be identified by a name or label.

Clearly the first possibility involves no problem of representation. The

12. Lorie, R. A., "XRM - An Extended (N-ary) Relational Memory," Technical Report 320-2096, IBM Scientific Center, Cambridge, Mass. (January, 1974).

second involves indicating the number of repetitions of a given group, and will be considered in the subsection on "Determination of a Field's Location". The remaining four possibilities are reasonable alternatives for representing records with variable logical contents.

The third alternative, an encoded control field, provides a representation with mathematically minimal storage space. However, it may require encoding and decoding procedures which are quite costly in terms of both time and storage. Accordingly, such optimal encoding, described below, will be used primarily to suggest and evaluate other techniques, described at the end of this subsection.

To introduce an optimal encoding scheme, assume that there are n optional fields. There are therefore 2^n possible combinations of fields present and absent. Each combination can be considered a "message" with some associated probability, say $p_i (1 \leq i \leq 2^n)$. Unless each optional field is present precisely half the time, these messages will occur with unequal probabilities. Therefore, it is reasonable to represent the more probable messages with short code words, and the less probable with longer code words. The following discussion merely provides a technique for optimally assigning code words. Communication theory[13] can be applied to yield the result that the average number of bits required to identify a message (i.e., to specify which fields are present) is bounded below by

$$-\sum_{i=1}^{2^n} p_i \log_2 p_i$$

This bound can be approached arbitrarily closely if arbitrarily large

13. Fano, R. M., "Transmission of Information", M.I.T. Press, Cambridge, Mass. (1961).

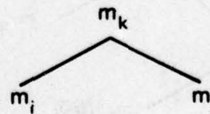
numbers of "messages" are coded together, i.e., if an arbitrarily large number of records can be considered as one super-record, with one code field indicating the presence or absence of all fields in all records. This is clearly infeasible, but simpler encodings will provide acceptable results. Huffman coding[13,14] will be described here and is also applicable to the data compression discussed in the subsection on "Determination of Field Values".

The following simple procedure works by constructing a binary tree, with the most probable messages near the root and the least probable the farthest from it.

Huffman Coding

1. Each message, with associated probability, is put onto a list.

2. The two messages with least probabilities, say m_i and m_j are removed from the list. A new message, say m_k, with the sum of those probabilities, is added to the list, and is represented as the following tree



3. If there is more than one message on the list, go to 2.

4. Otherwise, the result will be a binary tree. Assign a 0 to each left branch and a 1 to each right branch. A message at a leaf is then identified uniquely by the sequence of 0's and 1's which are on the path from the root to it.

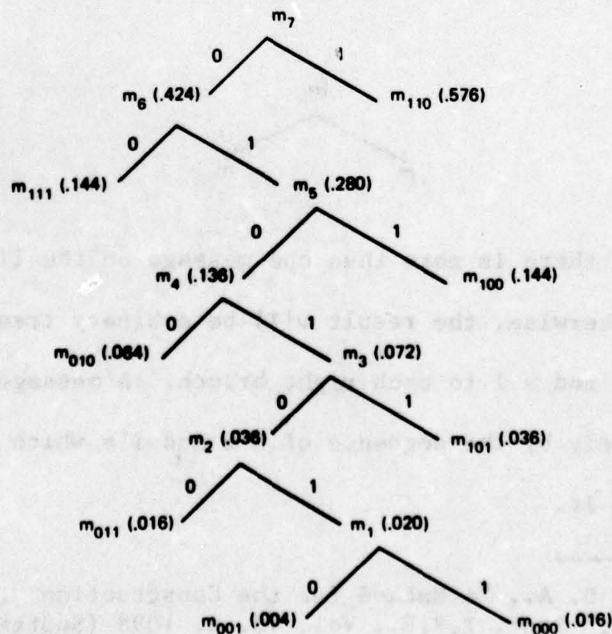
---

14. Huffman, D. A., "A Method for the Construction of Minimum-Redundancy Codes," Proc. I.R.E., Vol. 40, p. 1098 (September, 1952).

This procedure can be proven to yield an optimal encoding, in the sense that no other encoding of the same set of messages has a smaller average number of bits per message. The procedure will be illustrated by three fields with the following probabilities: $p_1 = .9$, $p_2 = .8$, $p_3 = .2$. The fields are assumed to be independent. The messages and their associated probabilities are given below:

$p(m_{000}) = .016$	(no field present)
$p(m_{001}) = .004$	(field 3 present)
$p(m_{010}) = .064$	(field 2 present)
$p(m_{011}) = .016$	(fields 2 and 3 present)
$p(m_{100}) = .144$	(field 1 present)
$p(m_{101}) = .036$	(fields 1 and 3 present)
$p(m_{110}) = .576$	(fields 1 and 2 present)
$p(m_{111}) = .144$	(fields 1, 2 and 3 present)

The corresponding coding tree is shown in the next figure, with probabilities in parentheses following the messages. The messages m_1 to m_7 were produced by combining other messages (e.g., m_1 means that either m_{001} or m_{000} occurred).



The following table gives the name of each message, its coding, the length of its coding, its probability, and the product of coding length and probability:

message	code	length	p	p x length
m_{000}	0101011	7	.016	.112
m_{001}	0101010	7	.004	.028
m_{010}	0100	4	.064	.256
m_{011}	010100	6	.016	.096
m_{100}	011	3	.144	.432
m_{101}	01011	5	.036	.180
m_{110}	1	1	.576	.576
m_{111}	00	2	.144	.288
				average length = 1.968

Hence, the average length of the code word is 1.968 bits, considerably better than the three bits required if each field were coded by a separate bit, and very close to the 1.913 bits required for the theoretical minimum (assuming many records were coded together).

Advantages and Disadvantages of Huffman Coding

Huffman coding, as noted earlier, has the advantage of providing the smallest average number of bits to represent the presence or absence of each field. The disadvantages are, first, that the encodings are variable length, which implies a need for variable length records, and, second, a possibly large amount of storage used by tables in the decoding process. The former disadvantage is common to the other techniques discussed below, so will not be a factor in choosing a technique. The storage problem is, however, quite serious: if there are n fields, then there are 2^n messages, each of which specifies the presence or absence of each of those n fields. Hence, the decoding tree consists of 2^n leaves of

n bits each. Disregarding the representation of the tree and the decoding procedure, this is $n2^n$ bits. A practical limit for such encoding, then, is probably around 10 fields (more than 10,000 bits). However, there are reasonable approximations to Huffman codings which are much less space consuming. Assume in the remainder of this subsection that the probabilities of occurrence of the fields are independent.

Any field whose probability, p, is near .5 can be encoded separately from the other fields by a single bit. The number of bits used is 1, so the difference in bits between this encoding and the optimal encoding is

$$1 + p \log_2 p + (1 - p) \log_2(1 - p)$$

bits.

The remaining fields can be divided into three groups: fields with probability near 0, fields with probability near 1, and everything else. If there are k fields with near 0 probabilities, say $p_1, \ldots, p_k$, then those fields which are present can be identified by labels of $[\log_2 k]$ bits each, where "[" and "]" brackets indicate the least integer greater than or equal to the real number inside them. A count field of $[\log_2(k + 1)]$ bits indicates the number of fields present. (An alternative scheme would be to use a special label to indicate the end of the fields, in which case both it and the other labels would require $[\log_2(k + 1)]$ bits.) The average number of bits used is

$$[\log_2(k + 1)] + \sum_{i=1}^{k} [\log_2 k] p_i$$

The average number of bits per field is

$$[\log_2(k + 1)]/k + \sum_{i=1}^{k} [\log_2 k] p_i/k$$

If the fields have equal probabilities, say p, then this expression reduces to

$$[\log_2(k + 1)]/k + [\log_2 k] p$$

This function has local minima for k of the form $2^j - 1$, where j is an integer. The following table shows the function for p = .1:

k	bits/field
7	.73
15	.67
31	.66
63	.70

(Clearly, there is no reason to use k = 63; two groups, each with k = 31, would be superior, even if one field had to be represented by a separate control field.) Note that these values compare reasonably well with the minimum possible value of .469. The values are much superior to the alternative of using a control bit for each field. A similar scheme can be used for fields whose probability of occurrence is near 1: labels are used for those fields which are not present.

An alternative representation for fields with probability near 1 may be the use of a special, null value, e.g., 0 in a pointer field, -0 in a 1's complement numeric field, or an illegal bit configuration in a character field. The expected number of bits is then the probability of the field being absent times the field length.

The remaining group of fields (with probabilities not near 0, .5, or 1) may be subdivided into small groups, each of which is represented separately by Huffman encoding. This procedure greatly simplifies the encoding/decoding trees: if a group of n fields is subdivided into m equal groups, the tree is reduced from 2^n leaves of n bits each to m trees of $2^{n/m}$ leaves of n/m bits each, or $n2^{n/m}$ bits total. As noted in the example of three fields, the encoding can be very good even for a small number of fields. The critical factor is that messages which are combined should be of about equal probability, because they are distin-

guished by a bit in the decoding tree, and a bit represents the maximum information if it distinguishes two equally likely messages. Equally probable branches are especially important at the top of the tree, where the probability of a message occurrence is large. Hence, the most likely message (i.e., the most likely combination of fields present and absent) should have a probability of .5 or less. This condition is fairly easily satisfied, since fields with near 0 and near 1 probabilities are put into separate groups. (For example, if three fields each have probability .8, then the most likely mesage has probability .512).

Summary Of Field Determination

The foregoing subsection has discussed Huffman coding as the optimal technique for determining fields which are present and absent in a record. Since Huffman coding for a large number of fields involves a very large encoding/ decoding tree, special cases are considered:

1. A field with probability of occurrence near .5 is determined by a single control bit, independently of other fields.

2. Fields with probability of occurrence near 0 are determined, if present, by labels in a control field or fields, plus a count to indicate the number of fields.

3. Fields with probability of occurrence near 1 are determined, if absent, by labels as above. They may also be determined by null fields.

4. The remaining fields are divided into small groups (the criterion for smallness being that the most likely combination of field occurrences should have probability .5 or less), which are encoded separately with Huffman coding.

Although this coding scheme is rather complex, it can be used to suggest and evaluate simpler alternatives.

DETERMINATION OF A FIELD'S LOCATION

A logical record may be represented by one, or by a number of physical records. In the latter case, a field's value may be obtained either by following a pre-defined access path to a record containing that field, or by invoking a procedure which computes the value, probably in a manner unknown to the data base management system. Conversely, a physical record may contain (parts of) many logical records.

Physical Record Corresponding to Logical Record

Consider first the case in whi.. all fields of a logical record are in one physical record. Then, in order to extract the field's value, it is necessary only to know the locations of the first and last bits of the field. If all fields are of fixed length and are always present in a fixed order, the required information can be stored in a data dictionary. If there are optional fields or if the order may vary, the data fields in a particular record must be described in a control field or fields. For example, a label or name, as in the previous subsection, could precede each field, or the order of the labels in contiguous control fields could indicate the identity and order of the data fields. A count field or a special label is necessary to indicate how many fields actually occur. Examples are shown below for physical representations of the following logical records:

EMP (ID#, NAME, PHONE)

001 JONES 12345

002 SMITH -----

If the order and field length are fixed, JONES's physical record might

be   001   JONESbbb   12345

(A "b" indicates a blank.) If the order is variable, then the record might be

I 001 N JONESbbb P 12345

where ID#, NAME, and PHONE are represented here by characters but would normally be encoded as, for example, 0, 1, and 2, respectively. If SMITH has no phone, and # (encoded as 3) is used to indicate end of record, then SMITH's record might be

N SMITHbbb I 002 #

Another representation is

I N # 002 SMITHbbb

where # indicates the end of the control fields. Another representation is

2 I N 002 SMITHbbb

where the initial control field is a count of the number of fields present.

If fields are not of fixed length, then it is necessary to know not only where a field begins but also where it ends. If the beginning of the first field is at a fixed location in the record, and the length of each field is stored within the record, then all the beginnings and endings can be computed. Or, if a pointer to the end of each field is stored, then the length of each field can be computed. For example, with the addition of a count field after the label indicating a name, SMITH's record can be represented as

2 I 002 N 5 SMITH

(the ID# is of fixed length, so needs no count). The count field functions as a pointer or offset from the first bit of a field to the first

bit of the next field. If a field has subfields - obviously the case if the field can be of variable length - then it is possible that an otherwise unused bit pattern in a subfield could be used as a flag to indicate the end of the field. For example, in the field with character subfields,

2 I 002 N SMITH!

it is assumed that "!" will never occur in an employee's name. Clearly the choice of count or flag depends on the possible length of the field. If a character is 6 bits, then a maximum of 63 characters can be described by a count of the same 6 bits.

The optimal representation for a count field is a Huffman code, where each "message" is a field length. For example, suppose that the following table gives the length, probability, and code for some field:

Length	p	Code
0	.001	1111110
1	.005	111110
2	.03	1110
3	.2	10
4	.7	0
5	.05	110
6	.01	11110
7	.004	1111111

The average length of the code is 1.485 bits, as compared with 3 bits for fixed length coding. As before, a Huffman code can be a very useful tool for evaluating the performance of other schemes; the savings of 1.515 bits might or might not be considered worth the extra processing in encoding and decoding.

If the order and size of the fields are assumed fixed, then the presence or absence of an optional field is indicated by one of the schemes of the preceding subsection. If the size of a field is variable, then a count field of 0 can be used to indicate that the field is absent, as in the example above. The cost of a 0 count field is (1-p)C, where p

is the probability that the field is present, and C is the size of the count field. This cost can be compared with those of the preceding schemes to determine the least costly representation for optional fields.

With labelling and a fixed order of ID#, NAME, and PHONE, the JONES and SMITH records are

001 N 5 JONES P 12345 #

and

002 N 5 SMITH #

assuming NAME and PHONE are both optional (note that the # in the first record is unnecessary; since all optional fields are present, the end of the record is known without it). With a count of 0 indicating a field which is absent, JONES's record is

001 5 JONES P 12345 #

and employee 003, whose name is unknown, has the record

003 0 P 67890 #

Other representations for optional fields were discussed in the preceding subsection.

Logical Record Contained in Many Physical Records

Assume now that the logical record need not be represented by contiguous fields. The possible alternatives and advantages are:

1. Frequently used or short fields may be placed in a primary record segment, with a pointer to less frequently used or longer fields which are placed in one or more secondary segments. Smaller segments of course, save transmission time and buffer space. Eisner and Severance[15]

15. Eisner, M. J., and D. G. Severance, "Mathematical Techniques for Efficient Record Segmentation in Large Shared Data Bases", Technical Report No. 261, Department of Operations Research, Cornell University, Ithaca, New York (July, 1975).

have devised an efficient algorithmic for computing optimal or near-optimal field assignments, given frequency of access to each field by each of a population of users. Access time, transfer time, the cost of storage, and the extra pointer required in segmentation are all considered in the algorithm.

2. Exceptional cases (such as very infrequently occurring fields, or fields which overflow normal maximum size) can be accommodated with little if any impact on normal cases, if the pointer is placed in the secondary segment. Bobrow discusses hashing schemes for efficiently finding the secondary segment from the primary.[16]

Binary Relations

The extreme case, in which all logical records are represented by binary relations, has four other advantages:[17]

1. Records may be of uniform size, simplifying each record access but adding to the number of accesses.

2. Only fields which are actually needed are accessed (the extreme of point 1).

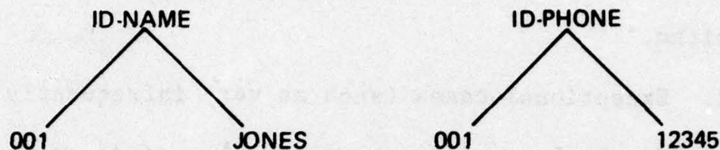
3. Fields may be added or deleted with no effect on the rest of the data base.

4. Extremely specialized access paths and redundant representations may be used to provide very fast response to complex queries.

The disadvantages of binary records are the many accesses necessary to retrieve a large logical record, and the cost in storage of linking records together.

16. Bobrow, D. G., "A Note on Hash Linking", CACM, Vol. 18, No. 7 pp. 413-415 (July, 1975).
17. Martin, J., "Computer Data-Base Organization," Prentice-Hall, Englewood Cliffs, New Jersey, p. 397 (1975).

Non-contiguous fields may be linked together in any of the ways discussed in the following section on physical access paths. For example to represent JONES's record, two binary relations might be:



where the identifier at the root of each tree is the relation name. Three of the possible ways of linking these two physical records are:

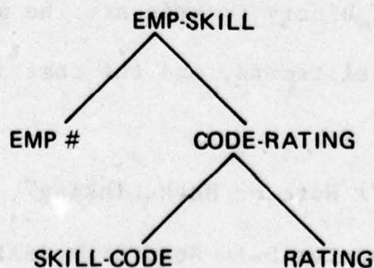
1. **An address pointer from ID-NAME to ID-PHONE,**
2. **An address pointer from ID-PHONE to ID-NAME,**
3. **The synonym chain, if the records are hashed on ID#.**

Linkages 2 and 3 have the advantage that an absent phone number consumes **no space. Furthermore, queries not involving phone number do not have to determine whether the field is present or absent; the encoding schemes of the preceding subsection generally involved some processing whether the optional field was needed or not.**

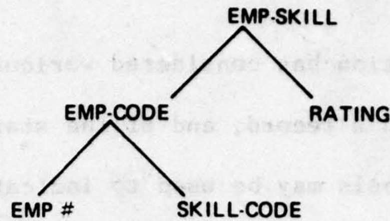
The representation is not so straightforward when the primary key consists of more than one field. ***For example,*** **the primary key of**

EMP-SKILL (EMP#, SKILL-CODE, RATING)

must be represented by two linked record types:



or



or another permutation of the leaves. Two accesses are now required to determine the RATING field value; one can selectively retrieve descriptive fields only after retrieving all of the primary key.

Physical Record Containing Many Logical Records

The preceding discussion pertained to fields located within a physical record corresponding to a logical record. Where possible, it is frequently desirable to store many identical fields only once - this reduces the cost of storing and updating the field, but may increase the cost and time to retrieve it. For example, in the logical record types

EMP (EMP#, NAME, ADDRESS)

EMP-SKILL (EMP#, SKILL-CODE, RATING)

the EMP-SKILL record type could be represented as a separate physical record or as a repeating group within EMP. In the latter case, EMP# would be stored only once within EMP, and not within each repetition of EMP-SKILL. Clearly, this facilitates the processing of queries such as "What are the skills of 001?" but complicates the processing of queries such as "Who has skill 3?"

For purposes of physical representation, repeating groups can be treated as individual fields. The number of repetitions, the location of each, the length of each, and the special case of no repetitions can all be represented by techniques already discussed in this section.

Summary Of Field Location

The preceding subsection has considered various representations of the order of fields within a record, and of the starting bit and the length of each field. Labels may be used to indicate both order and the fields which are present. A count field, possibly encoded by a Huffman code, or an otherwise unused combination of bits in a subfield, may indicate the length of a field. A 0 length may indicate the absence of an optional field.

The advantages and disadvantages of non-contiguous fields, and binary relations in particular, have been discussed. Binary relations can provide very fast response for retrieving a single field from a record identified by a single key field, but are costly in storage and unsuitable for retrieving a large number of fields.

DETERMINATION OF A FIELD'S VALUE

This subsection will consider various techniques for compressing the lengths of character strings. One technique, briefly discussed here, replaces character strings by pointers or index values. This may or may not reduce total storage requirements, depending upon how often character strings are repeated. A name file could be compressed very effectively, since a small number of first and last names are very common (Martin, 1975, pp.436-437).[17] This technique may increase processing speeds, since most manipulations (primarily comparisons) are performed on short, fixed-length fields. The cost of storage may also be reduced if the character strings can be placed on slow, cheap storage with only pointers in faster storage.

Two compression techniques, to be discussed in detail, involve elimination of repeated characters and character encoding (Martin, 1975, pp.433-448).[17] The two techniques differ in that the former is generally most effective when applied to a sorted collection of fields; the latter is generally limited to a field or even a single character within one record. The two techniques can be used simultaneously to provide very high degrees of compression on some types of data (especially sorted lists of names, key words, addresses and the like).

Consider first the elimination of repeated characters. Zeros and blanks frequently appear as leading or trailing characters in applications involving text processing or accounting. An obvious technique is to replace a string of such characters by a flag, such as an otherwise unused bit or character, and a count of the characters removed.

A much more effective technique, where applicable, is to sort all the occurrences of a given field (say NAME in the EMP record type of the preceding subsection) and eliminate repetitions of strings from one field to the next. For example, if a sequence of names is

SMITH, JOHN J.
SMITH, JOHN R.
SMITH, JOSEPH A.

then the compressed version is

SMITH, JOHN J.
12R.
9SEPH A.

where the numbers indicate the number of characters repeated from the previous field. Note that EMP records themselves need not be sorted by name. An effective technique is to have a sorted, compressed name file whose entries are retrieved by means of pointers or indexes from the EMP

records, which may then be sorted on EMP#. EMP# may obviously be easily compressed - e.g., by recording differences from the previous field. The critical requirement is, of course, that the field to be compressed be a sort field. This may be prohibitively expensive if that field is very volatile. The following section on physical access paths discusses various techniques for maintaining sorted records.

The second technique is the encoding of individual characters or very short and common sequences of characters. For example, Martin proposes a modification of the five-bit Baudot code to include three shifts: letter, number, and control (Martin, 1975, pp.440-442).[17] Since in many applications most strings are either alphabetic or numeric, but not both, shifts will be relatively rare so that the average character length will be about five bits, yet 87 characters are available (i.e., 32 bit combinations per shift minus 3 shift characters, or 29, times three shifts). As another example, in a name file, commonly used names and titles could be encoded by otherwise unused characters.

The most effective technique, and also the most complex, is again that of Huffman encoding. The modified Baudot coding can be improved to reflect the fact that alphabetic characters are far from equally probable in ordinary text. Even greater improvements may be made by encoding pairs or some triples of characters. The maximum improvement is dependent upon the frequencies of occurrence in a particular collection of fields, but for reasons of simplicity it may be preferable for one encoding scheme to be used for all suitable fields in a data base. In this case, encoding and decoding could be done very economically by means of one microprogram (Martin, 1975, p.447).[17]

As noted earlier, repeated characters may be eliminated first, and then the result encoded by some technique such as Huffman coding. Note further that measurement of the frequencies of occurrence of characters should be made on those characters actually encoded - i. e., the strings after elimination of repeated characters. In this case, the counts should be treated separately from the other characters, since their distribution is obviously quite different.

In summary, the elimination of repeated characters is simple and may be very effective in some instances. The elimination of repeated strings from sorted fields may be much more effective, but requires the maintenance of a sort order and a more complex process of encoding and decoding. Encoding of individual characters or short strings by Huffman coding can be extremely effective, but the encoding may be fairly costly in terms of time and the required coding tree.

SUMMARY OF PHYSICAL RECORD DESIGN

This section has concentrated primarily on various techniques to reduce mass storage requirements. The first subsection discussed techniques for indicating the presence and absence of optional fields. The second subsection discussed techniques for determining the location of variable-length or variable-order fields in a record. The final subsection discussed techniques for compressing character strings. The unifying technique of the section has been the applications of Huffman coding to compress records and to evaluate other techniques.

The following section, on the "Design of Physical Access Paths", discusses techniques for improving the speed of various operations, particularly retrieval.

DESIGN OF PHYSICAL ACCESS PATHS

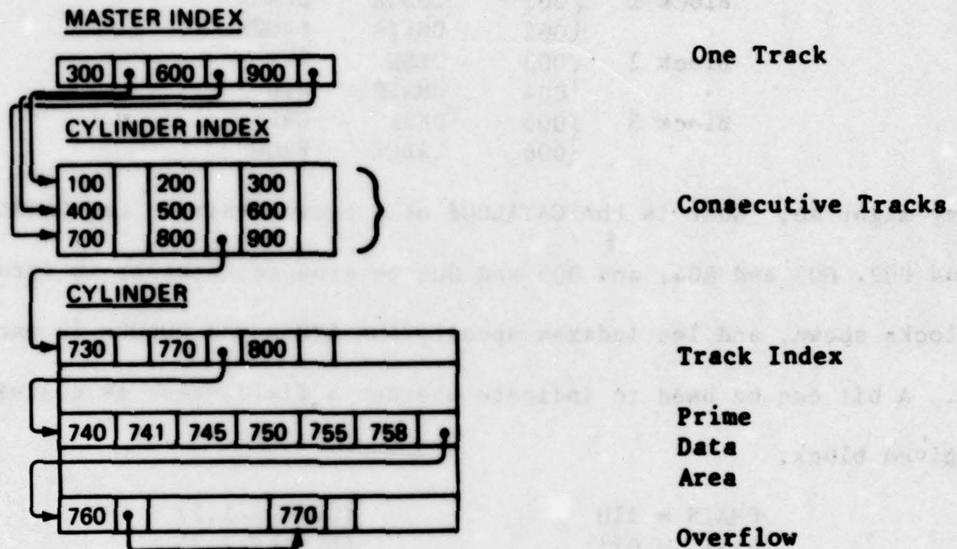
This section consists of five subsections. The first four discuss these topics: classification of the various types of physical access paths, examples of them, the parameters which apply to them, and the parameters which apply to their utilization. The final subsection is a summary.

The design of physical access paths will be treated as if it were independent of the design of logical and physical record structures, and independent of the design of logical access paths; that is, it is assumed that the data base management system is able to perform such functions as the amalgamation of physical segments of a logical record. The emphasis will be primarily on the efficiency of operation and only peripherally on such topics as the complexity and maintainability of the access programs. The assumption that any investment in those programs will be repaid by decreased response times or hardware requirements is reasonable only if the access program is heavily used.

BLOCKS

A physical access path is basically a method for partitioning the data base into smaller and smaller "blocks", where a block is either a record or a collection of blocks. For example, a search for a record using IBM's Indexed Sequential Access Method (ISAM) involves a search in a master index to find the appropriate track of the cylinder index, a search in that track for the appropriate cylinder, a search within that cylinder's index for the appropriate track, and finally a search within

the track for the appropriate record.[18] An example is given below (unnecessary details omitted):



Note that each entry in an index gives the last entry in the block. For example, to locate 760, the master index entry 900 is used (since 600 is less than 760), then the cylinder index entry 800 is used (700 is too small and 900 too large), then the track index 770. The track has overflowed, so the overflow chain is followed to 760.

A block is a hierarchical or tree structure in which the leaves consist of records. The highest level is directly accessable by the data base management system; lower levels are accessable only through movement within the tree. A record may be a part of many different blocks, each representing a different access path to it.

Complex queries - i.e., queries involving values for more than one field - can be made quite efficient by means of rather simple block

18. IBM Corporation, "Introduction to IBM Direct-Access Storage Devices and Organization Methods, GC20-1694-8", IBM, White Plains, New York, pp. 55-63 (1974).

structures. For example, suppose the data base consists of the following records:

FURNITURE	(CATALOG#,	TYPE,	COLOR)
Block 1	001	TABLE	BLACK
	002	CHAIR	BROWN
Block 2	003	DESK	RED
	004	CHAIR	RED
Block 3	005	DESK	GRAY
	006	TABLE	BROWN

A query might be, "What is the CATALOG# of a brown chair?" Let CATALOG#'s 001 and 002, 003 and 004, and 005 and 006 be grouped together to form the blocks shown, and let indexes specify the TYPEs and COLORs in each block. A bit can be used to indicate whether a field value is contained in a given block,

CHAIR	= 110	(blocks 1,2)
DESK	= 011	(blocks 2,3)
TABLE	= 101	(blocks 1,3)
BLACK	= 100	(block 1)
BROWN	= 101	(blocks 1,3)
GRAY	= 001	(block 3)
RED	= 010	(block 2)

A Boolean "and" of CHAIR and BROWN indicates that the only block which can satisfy the query is block 1. The number of records examined is minimized if the number of blocks is equal to the square root of the number of records (Martin, 1975, pp. 252-253).[17]

Block hierarchies which correspond to physical hierarchies are particularly important. For example, each step in the hierarchy from disk track, to cylinder, and finally to disk pack, mass storage cylinder or magnetic tape, represents an increase in access time of at least one order of magnitude. Processing time therefore increases greatly whenever a block is not contained within one of these storage units. A reasonable time at which to reorganize a block (e.g., to collect free

space) is when the block is about to overflow one of these physical boundaries. However, a block can sometimes be processed more quickly by dividing it into sub-blocks which can be processed independently - e.g., dividing a block among different disk volumes (Martin, 1975, pp. 210-213).[17] It may also be quite advantageous to divide a block according to activity of the sub-blocks - relatively inactive sub-blocks can be placed together on the outermost cylinders (which have slowest access) (Martin, 1975, pp. 212-214),[17] or even on slower devices. Different sub-blocks may also be organized differently, according to activity - e.g., the records of an active sub-block could be structured for fast retrieval while those of an inactive sub-block could be structured for storage efficiency. Knuth discusses various empirically common distributions of activity.[19]

In summary, then, simple access paths can be combined into a hierarchy of access paths, each level of which is a path to a smaller and more restricted subset of the data base. The access path through records at a given level may consist of one or more types of structures (described below). The design problem, then, is to determine an appropriate hierarchical structure and the parameters at each level. Detailed time and storage analyses have been made by Severance[20] and Yao,[21] and have been the foundations for computer programs which produce near-optimal physical access paths. More specialized programs have been produced for the

19. Knuth, D. E., "The Art of Computer Programming, Volume 3: Sorting and Searching", Addison-Wesley, Reading, Massachusetts, pp.396-399 (1973).

20. Severance, D. G., "Some Generalized Modeling Structures for Use in Design of File Organizations", PhD dissertation, the University of Michigan, Ann Arbor (1972).

21. Yao, S. B., "Evaluation and Optimization of File Organizations Through Analytic Modeling", Ph.D. dissertation, The University of Michigan, Ann Arbor (1974).

hierarchical IMS system[22] and the CODASYL DBTG network model.[23] An analysis by Senko et al[24] has been used as the basis for modeling and simulating a commercially available data base management system.[25]

Cardenas[26] has developed a model and program to determine storage costs and access times for tree, inverted file, and multilist structures assuming different query complexities and data base characteristics.

The following subsection will use the block concept as the basis for classifying the various types of physical access paths. The emphasis is on very simple alternatives: "data" vs "structure", "contiguity" vs "chaining", "ordering" vs "no ordering", "direct" vs "sequential" access. The objective is not elegance in the mathematician's sense of finding the simplest set of primitives, but rather to display all the choices which can be made. Such a low-level approach to the design of physical access paths will, it is hoped, help to isolate and correct those choices which are costly. This is important in any realistic environment, where changing patterns of data base usage require continual re-tuning of the physical structures.

---

22. Smith, S. E., and J. H. Mommens, "Automatic Generation of Physical Data Base Structures", Proc. 1975 ACM-SIGMOD International Conference on the Management of Data, ACM, New York, pp. 157-165 (1975).

23. Gerritsen, R., "A Preliminary System for the Design of DBTG Data Structures", The Wharton School, University of Pennsylvania, Philadelphia, (1975). Also published in CACM, Vol. 18, No. 10, pp. 551-557 (October, 1975).

24. Senko, M. E., E. B. Altman, M. M. Astrahan, and P. L. Fehder, "Data Structures and Accessing in Data-Base Systems", IBM Systems J., Vol. 12, No. 1, pp. 30-93 (1973).

25. Schneider, L. S. and C. R. Spath, "Quantitative Data Description", Proc. 1975 ACM-SIGMOD International Conference on the Management of Data, ACM, New York, pp. 169-185 (1975).

26. Cardenas, A. F., "Evaluation and Selection of File Organization - A Model and System", CACM, Vol. 16, No. 9, pp. 540-548 (September, 1973).

CLASSIFYING PHYSICAL ACCESS PATHS

Data and Structure

A block may contain "data" (i.e., representations of the logical data ultimately delivered to or obtained from a user) and "structure" (i.e., information, such as record addresses, which is unknown to the user, and which directs the traversal of the physical access path). The structural information may indicate, at a certain time, that a block which can potentially contain data is in fact empty - i.e., that the record contains "free space". Such records may be on the same physical access paths as those containing data, or may be on distinct access paths.

Pointers

A "pointer" from a record X of block A to a record Y of block B can be physically present in either X or Y, or in a separate structural block, C. If the pointer is in X, it can be a physical address, a data base key (i.e., an offset into a "translation table" of physical addresses), or a bit in a bit map (i.e., each record in B is assigned a position in a bit string, so a 1 in Y's position indicates that X points to Y). Access from X to Y involves no searching. If the pointer is really a "back pointer" in Y, the same possibilities exist, except that in this case B is searched to find the record (or records) which point back to X. This can be of great value if A is frequently accessed and resides on expensive storage, while B is infrequently accessed and resides on cheaper storage, since the pointer is on the cheaper device and does not have to be read when records in A are read. If the pointer is in C, then it actually consists of two pointers - one to X and one to Y. It could

be a collection of pairs of addresses or data base keys, ordered on one or both blocks, or a two-dimensional bit map. Data compression techniques can be employed on bit maps or on collections of address pairs. In the remainder of this section, a pointer is generally assumed to be either an address or data base key in X. Other possible implementations should be considered in special cases. For example, an encoded bit map may be a very efficient implementation of an index of records with a particular field value.

Contiguity and Chaining

Blocks may be combined into higher-level blocks by contiguity (i.e., the address of the i+1-th block of the collection is equal to the address plus the length of the i-th block) or chaining (i.e., the i-th block of the collection points to the i+1-th block). The traversal of a physical access path involves two types of movement: one from block to block of a collection until an appropriate block is found (i.e., horizontal tree traversal); the other a search within that block (i.e., movement down the tree). A block may be constructed using both contiguity and chaining - e.g., the records of an ISAM track are contiguous, with the last record possibly beginning a chain of overflow records. A block may also consist of different types of sub-blocks - the leaves (records) may occur at different levels in the tree constituting the sub-block.

Ordering, Direct Access, and Sequential Access

If sub-blocks within a block have known addresses (e.g., all addresses are known if the sub-blocks are of fixed length and contiguous) and are ordered according to a certain field, then a search for a sub-block with

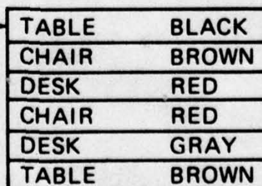
a particular value for that field can be performed in three different ways. First, it may be possible to directly access the desired sub-block - i.e., the field value is transformed by a 1-to-1 function into the address of the sub-block or some other sub-block from which it can be reached. Second, it may be possible to access a sub-block which is either that desired, or which can begin another search - i.e., the value is transformed by a many-to-one function into the addresses of the desired sub-block or some other sub-block from which it can be reached. Third, sub-blocks can be accessed sequentially until the desired sub-block is found. Access must be sequential if the addresses of sub-blocks are unknown or if the sub-blocks are not ordered on the appropriate field.

The following subsection will present examples of different physical access paths which can be constructed by various combinations of different kinds of block structures.

EXAMPLES OF PHYSICAL ACCESS PATHS

Sequential

A sequential file - e.g., a tape file - is a block of contiguous data records, for example:

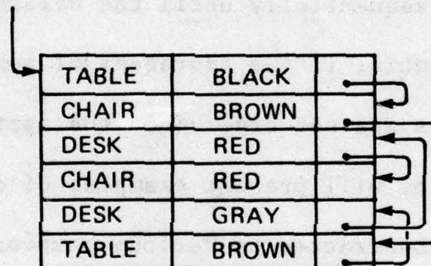


The arrow indicates that the address of the first record is known. Ordering may reduce retrieval time but increases maintenance time. For example, retrieval time for all items of a certain color is approximately one-third as great for an ordered as for an unordered tape file

(assuming that color values are uniformly distributed in the requests, and that the tape can be read forwards or backwards from any point within the tape); however, inserting a new record involves copying the entire tape. The trade-off between retrieval and maintenance costs is a general characteristic of physical access paths.

List

A list is a chained block of data records, for example



This list is ordered on color and unordered on type of furniture. The objective of ordering is to reduce retrieval time, but the cost of maintenance is much more moderate than with a sequential tape file. Insertion requires only an additional search for the proper predecessor record plus access to a record in free space.

Multilist

An index is an ordered contiguous block of structure records. The diagram below shows two indexes to a multilist,[27,28] plus the data records themselves:

---

27. Lefkovitz, D., "File Structures for On-Line Systems", Hayden, Rochelle Park, New Jersey, pp. 126-129, 157-165 (1969).
28. Lefkovitz, D., "Data Management for On-Line Systems", Hayden, Rochelle Park, New Jersey, pp. 62-64, 118-120 (1974).

TYPE INDEX

2	CHAIR
2	DESK
2	TABLE

DATA

TABLE	001	BLACK
CHAIR	006	BROWN
DESK	009	RED
CHAIR	002	RED
DESK	007	GRAY
TABLE	003	BROWN

COLOR INDEX

BLACK	1
BROWN	2
GRAY	1
RED	2

The numbers indicate the lengths of the lists. Each index entry is the address of the beginning of a chain of records with the indicated field value. Note that each chain is in ascending order of physical address; the reason for this will be discussed in the following paragraphs. The maintenance of such a chain is more complex than for an unordered chain. If space for new records is allocated from the high end of free space, then a new record will have an address lower than previously allocated records, placing it at the beginning of the chains; periodic reorganization is necessary to move new free space, resulting from deletions, to the lower addresses. Otherwise, each chain must be followed to find the proper place to insert the new record; hence, inserting into a multilist of n chains requires n searches for the proper locations in order to maintain ascending order. To satisfy a complex query such as, "What is the CATALOG# of a brown chair?", the shorter of the BROWN and CHAIR lists will be followed (in this case, they are equally long), checking each record for the other field value.

If ascending order is maintained, retrieval may be much faster, since cylinder access is minimized. A detailed analysis is given in the following paragraph.

Analysis of Time for Multilist and Sequential Retrieval

A multilist organization which is not ordered by ascending physical address may actually be less efficient in retrieval than a sequential file, since it is possible to consume more time accessing a few records in random locations than all records sequentially. Assume, for example, that k records are retrieved from a disk file stored on M tracks. Assume further that the time for one rotation is R, that the time to move to an adjacent cylinder is C_1, that the time to move over j cylinders, for j>>1, is $C_2+j\ C_3$, and that there are n tracks per cylinder. Then, ignoring positioning to the first record, the time to retrieve k records by following a chain of randomly distributed addresses is

$$T_R \cong (k-1)\ (C_2 + C_3 m/3n + R/2)$$

Assuming that pm tracks contain at least one record, and that at least one record is contained on each cylinder, the time to retrieve the k records by following a chain of ascending addresses is

$$T_A \cong (m-1)\ C_1/n + (pm-1)\ R$$

An equation for p will now be derived. Assuming that there are b records per track, there are therefore bm records, of which k are to be retrieved. The probability of retrieving any given record is

$$k/bm$$

Hence, the probability of not retrieving a given record is

$$1 - k/bm$$

and the probability of not retrieving any record from a track is

$$(1 - k/bm)^b$$

Hence,

$$p = 1-(1-k/bm)^b$$

$$= k/m - k^2(b-1)/2!m^2b + k^3(b-1)(b-2)/3!m^3b^2 - \ldots$$

For $k << m$,

$$pm \cong k$$

so that

$$T_A \cong (m-1)\ C_1/n + (k-1)\ R$$

The time to sequentially retrieve all records from all m tracks is

$$T_s \cong (m-1)\ C_1/n + m\ R$$

For an IBM 2314 disk drive, $n=20$, $R \cong 25$ ms (milliseconds), $C_1 \cong 25$ ms, $C_2 \cong 50$ ms, and $C_3 \cong .4$ ms (IBM, 1974, p.22).[18] Hence, in milliseconds,

$$T_R \cong (k-1)\ (62.5 + .007m)$$

$$T_A \cong 1.25\ (m-1) + 25\ (k-1)$$

$$T_S \cong 26m$$

The following table gives approximate values of T_R, T_A, and T_S in seconds for m=2000 tracks and the indicated values of k:

K	T_R	T_A	T_S
100	7	5	52
500	38	15	52
700	53	20	52
1000	76	23-27*	52

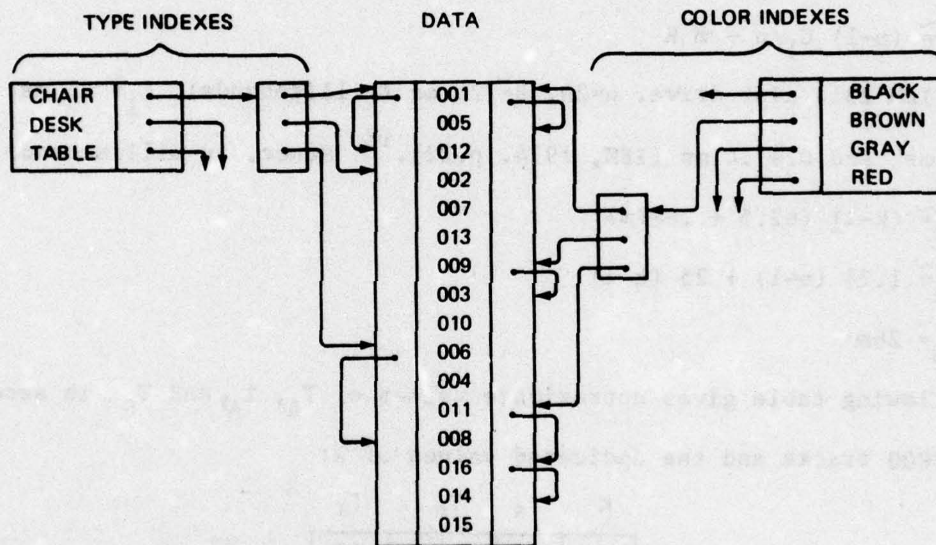
For larger values of k, T_A begins to approach T_S, while T_R continues to grow linearly. For smaller values of k, T_A and T_R are approximately equal. This analysis, of course, ignores CPU time and channel activity, which could be significant for the sequential organization. However, it

*This entry depends on b, the number of records per track, since the condition $k << m$ does not hold. The value is 23 for large b (i.e., small records) and 27 for b=1.

does serve to illustrate the points that ordering by increasing address is generally advantageous, and that sequential organization may be reasonable even for a rather low "hit-ratio" (density of records to be retrieved).

Bounded Multilist

A second level of index may be used to bound the lengths of the chains in the multilist. For example,

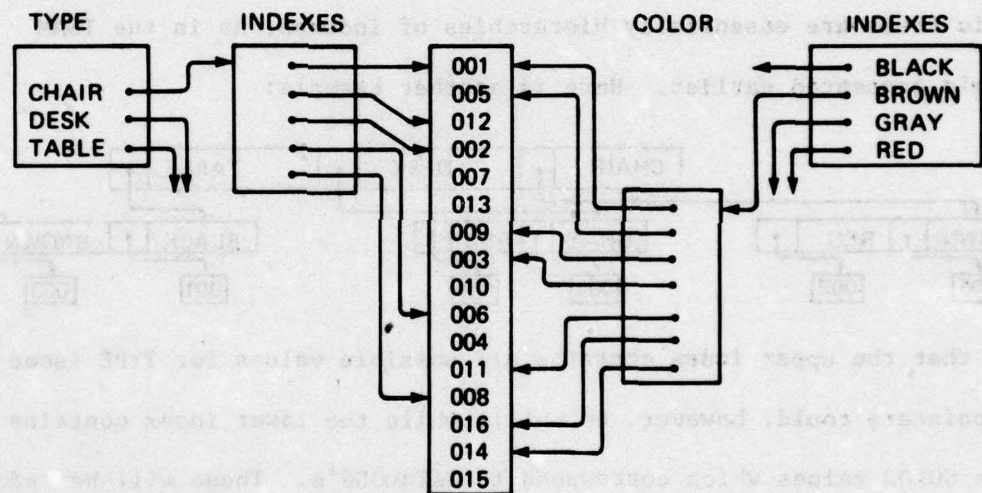


For clarity, some pointers are not shown. Note that the TYPE and COLOR fields are omitted. A complex query such as that introduced earlier in the section, "What is the CATALOG# of a brown chair?", can be more efficiently satisfied if both the starting and ending addresses of the chairs are indicated in the index: in this case only the first and third TYPE chains need be examined. Note that both levels of index are relatively stable compared with the data. Insertions and deletions in the data will only rarely result in insertions or deletions in either index, so that contiguous organizations are appropriate. Bounding the chains may greatly

facilitate maintenance of the data records, since a short chain will be followed to determine the proper place of insertion. If each chain is restricted to a single "cell" (i.e., a physically significant block, such as a cylinder), then the analysis at the beginning of this subsection applies, and access time can be greatly reduced.

Inverted File

If the multilist chains are restricted to single records, the result is an inverted file. For example,

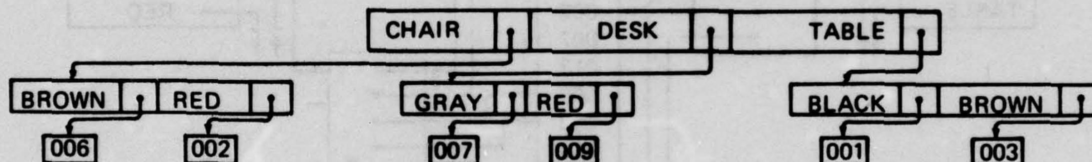


The complex query referred to earlier, "What is the CATALOG# of a brown chair?", can now be reduced to a comparison of two lists (ordered in this case) in the indexes and one reference to a data record. The total number of pointers is approximately the same as for either the multilist or bounded multilist, but now all pointers are in an index block, rather than in a data block. This is desirable if some fields are used in more queries than other fields, since they can be placed on faster storage devices. The primary disadvantage of the inverted file is that

maintenance is costly - each index must be updated every time a record is updated. In the case of the bounded multilist, update could easily be deferred, since the only effect of a chain growing beyond the optimal size is a small degradation in performance.

TRIE and TREE

Tree-structured blocks can be either static (i.e., the number of levels is fixed, although the contents of the nodes may vary) or dynamic (i.e., the shape of the tree varies with insertions and deletions). Static trees are essentially hierarchies of indexes, as in the ISAM example presented earlier. Here is another example:



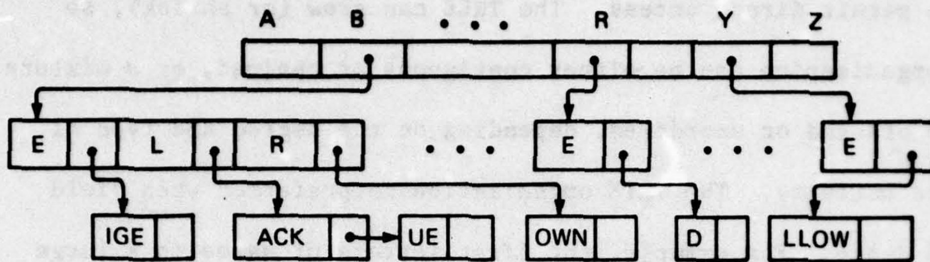
Note that the upper index contains all possible values for TYPE (some of the pointers could, however, be null), while the lower index contains only those COLOR values which correspond to CATALOG#'s. These will be referred to as TRIE[29] (Pronounced "try") and TREE[30, 31] structures, respectively. The decision of when to use TRIE and when to use TREE is based on storage costs (the TREE is a compressed TRIE, so the analysis of the section on Design of Physical Records is applicable), and on retrieval and maintenance costs. Note that the TRIE does not grow, so should be ordered and con-

29. Fredkin, E., "TRIE Memory", CACM, Vol. 3, pp. 490-499 (September 1960).

30. de la Briandais, R., "File Searching Using Variable Length Keys", Proc. 1959 Western Joint Computer Conference, IEEE, New York, pp. 295-298 (1959).

31. Severance, D. G., "Identifier Search Mechanisms: A Survey and General Model", ACM Computing Surveys, Vol. 6, No. 3, p. 186 (September 1974).

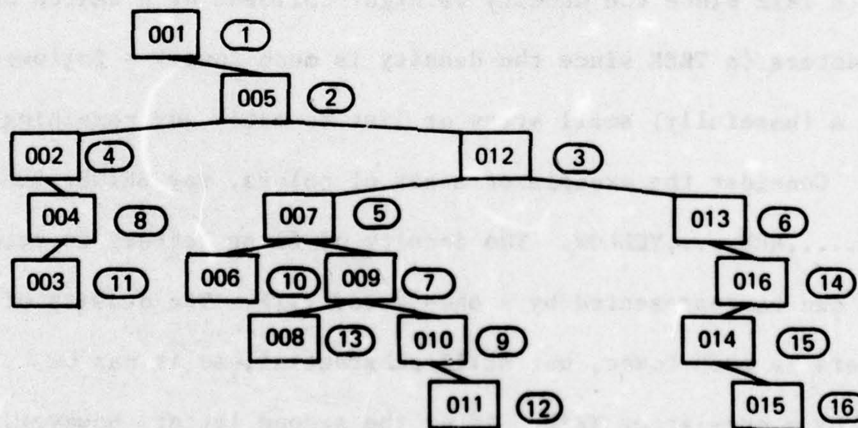
tiguous to permit direct access. The TREE can grow (or shrink), so that its organization can be either contiguous or chained, or a mixture, and either ordered or unordered, depending on the degree and type of maintenance activity. The TRIE organization is preferred when field values are dense. For example, the first letters of names in a large telephone directory are dense - all letters occur at least once. However, the TREE organization is preferred when field values are sparse. For example, the fifth letters of names in a telephone directory are extremely sparse - very few different letters can occur after four letters have already occurred. Severance[20,31] has developed a generalized character-string search model, described further at the end of this subsection, which exploits this rather common phenomenon. The first steps in his generalized search are two prefix searchs - i.e., a search on the first i characters (a TRIE since the density is high) followed by a search on the next j characters (a TREE since the density is much lower) - followed by a search of a (hopefully) small array or list to match any remaining characters. Consider the example of a set of colors, say BEIGE, BLACK, BLUE, BROWN,...,RED,...,YELLOW. The density of first letters is rather high, so it can be represented by a one-letter TRIE. The density of second letters is much lower, but still substantial, so it can be represented by a one-letter TREE. After the second letter, however, there is generally only one color, so a list is suitable for handling synonyms from the TREE. The following diagram shows the result, where the letters are above the TRIE records, to indicate that, since all characters are present, they are implicit and need no storage space. Pointer spaces which are empty indicate that no entry exists - e.g., no color (in this set) begins with A.



Note the decisions which have been made: one character in the TRIE search, which maps into a *TREE*, one character in the TREE search, contiguity rather than chaining in the TREE search, and chaining in the final list.

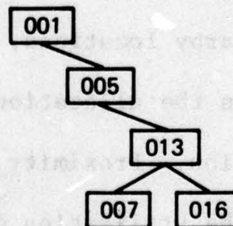
Search Trees

Search trees (Knuth, 1973, pp. 422-499)[19] are tree-structured blocks *which can grow vertically, for example*, the following unbalanced binary search tree:

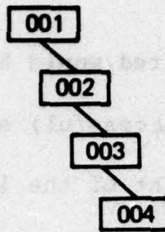


Unless otherwise noted, all search trees to be discussed are binary. The circled numbers indicate the order of insertion into the tree. Given a value, say 007, search begins at the root. If the value is not at that node, search proceeds with the left or right subtree as the value at the node is less than or greater than the search value.

Hence, the sequence of nodes visited would be 001, 005, 012, and finally 007. Insertion involves an (unsuccessful) search followed by linkage of the new record to the left or right of the last node reached - e.g., 017 would be to the right of 016. Deletion is simple if the node to be deleted is a leaf or has only one descendant, but is somewhat complicated if the deleted node has two descendants (Knuth, 1973, pp. 428-429).[19] In the latter case deletion involves replacement of the node by the largest node in its left subtree or the smallest node in its right subtree. For example, 012 can be deleted by replacing it with either 011 or 013. Node 011 cannot have a right descendant, since that would be greater than 011, and 011 was chosen as the largest in the left subtree of 012. Similarly, 013 cannot have a left descendant. After replacing 012, the old 011 or 013 node is removed, and its subtree (if any) linked into the tree. The result of replacing 012 by 013 would be as follows, where unchanged subtrees have not been shown:



Note that the search tree is poorly balanced - i.e., some paths from root to leaf are much longer than others. Search of a perfectly balanced binary tree of 2^K-1 nodes (K levels) requires access to about K nodes. However, $(2^K-1)/2$ nodes are required to search a tree in which there is a single leaf and all other nodes have one descendant, as in this tree:



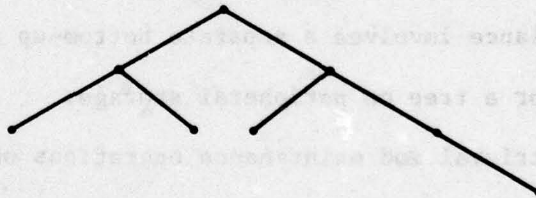
Knuth (1973, pp. 426-427)[19] shows that, for random insertions, the expected number of nodes accessed in a search is about 1.4K, which may or may not be acceptably close to the minimum, K. However, insertions come in order in many practical cases. For example, insertion of new part number, say 001, may be followed by the insertion, in order, of 001A, 001B, ..., 001N - the result will be a list of maximum length 14 and average search length 7, rather than a balanced tree with maximum 4 and average search length 2. If the search cost is determined by the search length, this degree of imbalance could be unacceptable. However, if the tree is on a peripherial storage device, then nodes 001A, 001B, ..., 001N would probably be placed in nearby locations, since they were inserted in sequence. Proximity depends on the allocation of free storage, to be considered in the next subsection. Proximity would tend to make the search fast and efficient in the utilization of input-output channels, since the number of track and cylinder accesses and the number of records transferred are more significant than the number of nodes examined.

If a tree is to be maintained in approximate balance, there are three alternatives. First, insertions, modifications, and deletions can be collected into batches and ordered (e.g., randomly, to try to eliminate long lists) before maintenance is performed. Second, a

reasonable precaution, even given the first alternative, is to periodically reorganize the data base (e.g., when it is not available to on-line users). Third, the tree may be examined during each insertion, deletion, or modification and reorganized if necessary to ensure approximate balance.

The last alternative, maintenance of approximate balance, has received a great deal of attention. Nievergelt, in a survey of the field,[32] classifies approximately balanced trees as either "height-balanced" or "weight-balanced".

A height-balanced tree, as exemplified by AVL trees*, satisfies the condition that, at any node, the heights of the left and right subtrees differ by at most some constant (for example, 1 for AVL trees). Thus, the following is a height-balanced tree:



Search of a height-balanced tree is slightly longer than that of a completely balanced tree, but reorganization is much simpler. The maximum search length in a completely balanced tree of n nodes is $\log_2(n + 1)$, and in a height-balanced tree is $1.44 \log_2(n + 1)$; the average search lengths differ by a constant.[32] More general height-

---

32. Nievergelt, J., "Binary Search Trees and File Organization", ACM Computing Surveys, Vol. 6, No. 3, pp. 195-207 (September 1974).

*So-called after the originators Adel'son-Vel'skii and Landis. Described in Knuth, 1973, pp. 451-468.

balanced trees are described by Foster.[33] A perfectly height-balanced but non-binary "B-tree" is described by Bayer; this tree has the interesting property that it grows horizontally as long as branches are available, and then grows a new root[34,35,36] (Knuth, 1973, pp. 473-479).[19]

A "weight-balanced" tree, as exemplified by the BB-tree of Nievergelt,[37] has the property that, at any node, the ratio of the number of leaves in the left and right subtrees is bounded above and below. The tree can be more or less balanced, depending on the bounds, and thus provide any desired compromise between retrieval and maintenance speeds. Maintenance of weight balance seems more efficient than that of height balance (though it may involve more operations), since weight imbalance can be corrected at the node at which it is detected: height imbalance at the root may not be detected until a leaf is reached. Hence, correction of height imbalance involves a separate bottom-up pass, which could be time consuming for a tree on peripheral storage.

In general, retrieval and maintenance operations on approximately balanced trees of n nodes require access to a number of records which is on the order of log n. Unbalanced trees may require access to a

---

33. Foster, C. C., "A Generalization of AVL Trees", CACM, Vol. 16, No. 8, pp. 513-517 (August 1973).

34. Bayer, R. "Binary B-trees for Virtual Memory", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM, New York, pp. 219-235 (1971).

35. Bayer, R., "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", Acta Informatica, Vol. 1, No. 4, pp. 290-306 (1972).

36. Bayer, R., and E. M. McCreight, "Organization and Maintenance of Large Ordered Indexes", Acta Informatica, Vol. 1, No. 3, pp. 173-189 (1972).

37. Nievergelt, J., and E. M. Reingold, "Binary Search Trees of Bounded Balance", SIAM J. Computing, Vol. 2, No. 1, pp. 33-43 (March 1973).

maximum number of records on the order of n, so might appear less attractive. However, efficient maintenance of balance requires extra storage for data about the relative balance. This extra storage reduces the number of nodes which can be read in one access from a peripheral device, and hence increases the number of accesses.

Held and Stonebraker[38] argue that static trees with overflow, such as ISAM, may be preferable to B-trees since they require fewer pointers and hence fewer accesses. An example demonstrates that periodic reorganization costs less than B-tree maintenance. However, the greater cost of maintenance may not be as important as a guarantee that the average or maximum retrieval time will not exceed a certain value; some form of balanced tree would be required to provide such a guarantee.

Hashing

Hashing, or key transformation, is a technique for mapping a large collection of possible record identifiers onto a much smaller number of addresses of "buckets".[31,39,40] Each bucket consists of a fixed number of "slots", and each slot can hold one record. For example, hashing is frequently used in building symbol tables for compilers or assemblers. The number of possible variable names is extremely large - there are 26 X 36^5 possible six-character names in FORTRAN, for example - but the

38. Held, G., and M. Stonebraker, "B-trees Re-examined", Memorandum No. ERL-M528, Electronics Research Laboratory, University of California, Berkeley (2 July 1975).
39. Maurer, W. D., and T. G. Lewis, "Hash Table Methods", ACM Computing Surveys, Vol. 7, No. 1, pp. 6-19 (March 1975).
40. Peterson, W. W., "Addressing for Random-Access Storage", IBM J. Research and Development, Vol. 1, No. 2, pp. 130-146 (April 1975).

number of actual names in even a very large program is comparatively small - say a few hundred or thousand. If a function can be found which maps the actual names "uniformly" into the set of bucket addresses - i.e., the number of names mapped into each bucket is approximately the same for all buckets - then any particular name can be found in about one access. Some provision must be made for handling "collisions" - too many records mapping into one bucket, causing "overflow." Hence, insertion of a record involves two processes: mapping the record into a "home" bucket, and, if that bucket has no free slot, "probing" (following a sequence of overflow addresses) until free space is found.

The following paragraphs discuss different mapping functions, methods for handling overflow, and the advantages of different bucket sizes. Note that a sequence of overflow records is really a (hopefully small) block - hence, design of an overflow record is really another problem in physical access path design. Two subproblems are involved: one to determine a free storage area, and the other to link the overflow records together. A bucket is, of course, a level in the hierarchy of blocks - the organization of records within a bucket must be determined.

The mapping function from the set of identifiers to the set of bucket addresses is clearly quite important - a bad function will cause a large number of collisions, with long overflow sequences. Fortunately, a very simple function - division hashing, or using the identifier modulo the number of buckets - proved to be generally quite good in real appli-

cations studied by Lum, Yuen, and Dodd.[41,42] Division hashing was, in fact, better than a randomizing function. That is, a random assignment of addresses to identifiers does not exploit the regularities in real data, which are exploited, to some degree, by division hashing. Buchholtz[43] and Knuth (1973, pp. 508-513, 521-524, 542-543)[19] also discuss other mapping functions which may be useful in special situations.

The two most common methods of overflow sequencing are "open" and "chained". An open sequence is generated by a fixed series of modifications to the original hashing function - e. g., in linear probing the n+1-th address to be examined is simply the first plus a multiple of n, modulo the size of the prime area. Open overflow has the advantages of simplicity and storage economy. However, if the previous simple sequence of overflow addresses is generated, overflow sequences may become quite long through a process of "clustering" of sequences of different "synonyms" (records mapping into the same prime bucket). For example, using division hashing with a prime area size of 7 and a bucket size of 1, and linear probing with constant 1, the insertion sequence 1, 8, 2, 9 produces the following result:

0	1	2	3	4	5	6
	1	8	2	9		

(Bucket numbers are shown above the buckets.) Inserting 15 involves

---

41. Lum, V. Y., P. S. T. Yuen, and M. Dodd, "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files", CACM, Vol. 14, No. 4, pp. 228-239 (April 1971).

42. Lum, V. Y., and P. S. T. Yuen, "Additional Results on Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files", CACM, Vol. 15, No. 11, pp. 996-997 (November 1972).

43. Buchholz, W., "File Organization and Addressing", IBM Systems J., Vol. 2, pp. 86-111 (June 1963).

following the bucket sequence: 1, 2, 3, 4 and finally insertion in bucket 5. Note that 1 and 8 are synonyms, 2 and 9 are synonyms, and the sequences of synonyms have run together to form a long clustered overflow sequence. This phenomenon can be avoided by a more complex overflow sequence. For example, in quadratic probing,[44,45] the n-th address in the sequence of synonyms is the first address plus a quadratic function of n, say $(n-1)^2$, modulo the size of the prime area. The previous example would be:

0	1	2	3	4	5	6
	1	8	2			9

and 15 would be inserted in bucket 5 after following the bucket sequence 1, 2. The overflow sequences no longer coalesce. A possible disadvantage is that only one-half the bucket addresses will be generated before the sequence repeats, given that the number of buckets is a prime. This may not be significant, since the overflow sequence is then so long that reorganization is necessary. Techniques have been developed, however, for searching the entire table[46] (Radke, 1970, p. 104).[45] Of particular importance in many applications are numbers of buckets which are powers of two; full-table searching is possible with appropriate choice of the

44. Maurer, W. D., "An Improved Hash Code for Scatter Storage", CACM, Vol. 11, No. 1, pp. 35-38 (January 1968).

45. Radke, C. E., "The Use of Quadratic Residue Research", CACM, Vol. 13, No. 2, pp. 103-105 (February 1970).

46. Day, A. C., "Full Table Quadratic Quotient Searching", CACM, Vol. 13, No. 8, pp. 481-482 (August 1970).

quadratic.[47,48,49] "Secondary clustering", the phenomenon by which actual synonyms map into the same sequences of overflow addresses, can also be eliminated by making each new overflow address depend on the identifier, rather than on its hashed value.[50,51,52,53]

The average number of buckets accessed to locate a record using open overflow with various bucket sizes is given below for various "loading factors" (the number of records divided by the number of slots) (Knuth, 1973, p. 535):[19]

Bucket Size	Loading Factor .70	.80	.90	.95
1	2.2	3.0	5.5	10.5
2	1.5	1.9	3.1	5.6
5	1.1	1.3	1.8	2.7
10	1.04	1.1	1.3	1.8
20	1.01	1.04	1.1	1.4
50	1.001	1.005	1.04	1.1

Note that the numbers are for bucket accesses, not record accesses - hence, the advantage of large bucket size is not as great as would appear from the table. For example, if 20 records are read in one disk access, say a page,

---

47. Batagelj, V., "The Quadratic Hash Method When the Table Size Is Not a Prime Number", CACM, Vol. 18, No. 4, pp. 216-217 (April 1975).

48. Ackerman, A. F., "Quadratic Search for Hash Tables of Size p^n", CACM, Vol. 17, No. 3, p. 164 (March 1974).

49. Hopgood, F. R. A., and J. Davenport, "The Quadratic Hash Method When the Table Size Is a Power of 2", Computer J., Vol. 15, No. 4, pp. 314-315 (1972).

50. Bell, J. R., and C. H. Kaman, "The Linear Quotient Hash Code", CACM, Vol. 13, No. 11, pp. 675-677 (November 1970).

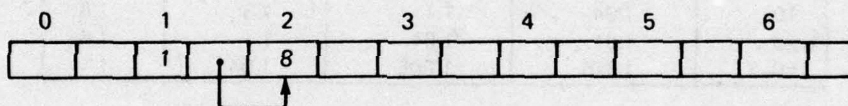
51. Lamport, L., "Comment on Bell's Quadratic Quotient Method for Hash Code Searching", CACM, Vol. 13, No. 9, pp. 573-574 (September 1970).

52. Bell, J. R., "Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering", CACM, Vol. 13, No. 2, pp. 107-109 (February 1970).

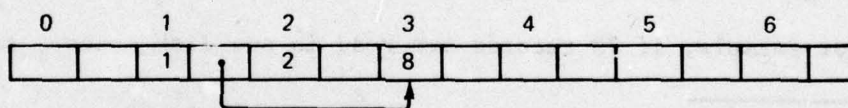
53. Burkhard, W. A., "Full Table Quadratic Quotient Searching", Computer J., Vol. 18, No. 2, pp. 161-163 (1975).

then a bucket size of 1 gives the same performance as bucket size of 20 if the overflow sequence first generates all slots on the page. Records should be small, so that many can be read simultaneously. Hashing with open overflow is therefore suitable for indexes, but not for large records, unless storage is sufficiently cheap that a low loading factor is feasible.

A chained overflow sequence is an ordinary chain of buckets in either the "prime" area - that addressed by the hashing function - or a separate overflow area. In the former case, overflow records can be moved about so that each chain consists only of records mapping into a single home bucket. For example, if division hashing is used with 7 buckets of size 1, then the result after inserting 1 and 8, with linear probing, is



and after inserting 2 is



The effect is to avoid clustering - overflow from one bucket causing, or increasing, overflow from another bucket, as discussed earlier. Clustering will also be avoided when using a separate overflow area if the bucket size is 1, or by restricting each overflow bucket to be chained to a single home bucket, or by moving records, as above.

The following table gives the average number of buckets accessed to locate a record, assuming chained overflow to a separate area with bucket size 1 (Knuth, 1973, p. 535):[19]

Bucket Size	Loading Factor .70	.80	.90	.95
1	1.4	1.4	1.5	1.5
2	1.2	1.3	1.4	1.4
5	1.1	1.2	1.3	1.3
10	1.06	1.1	1.2	1.3
20	1.02	1.06	1.2	1.2
50	1.001	1.02	1.1	1.2

Severance and Duhne,[54] and van der Pool,[55,56,57] present detailed analyses of the effects of different overflow techniques, bucket sizes, and loading factors on over-all system performance. As noted earlier, the open overflow technique is generally suitable only for small records. This technique is quite sensitive to the loading factor, and therefore is not suitable if the number of records varies greatly. It does have the advantage that overflow records tend to cluster close to the home bucket, thereby reducing track and cylinder accesses. The chained technique with separate overflow is suitable for large records or large buckets, since in either case the overhead of the chaining pointer is negligible. This technique is also relatively insensitive to the loading factor, so is suitable for volatile files. Severance and Duhne (1974, p.13)[54] observe that separate overflow buckets should be of size 1 unless speed of retrieval is much more significant than storage costs.

---

54. Severance, D. G., and R. Duhne, "A Practitioner's Guide to Addressing Algorithms: A Collection of Reference Tables and Rules of Thumb", Technical Report No. 240, Department of Operations Research, Cornell University, Ithaca, New York (November 1974).

55. van der Pool, J. A., "Optimum Storage Allocation for Initial Loading of a File", IBM J. Research and Development, Vol. 16, No. 6, pp. 579-586 (November 1972).

56. van der Pool, J. A., "Optimum Storage Allocation for a File in Steady State", IBM J. Research and Development, Vol. 17, No. 1, pp. 27-38 (January 1973).

57. van der Pool, J. A., "Optimum Storage Allocationn for a File with Open Addressing", IBM J. Research and Development, Vol. 17, No. 2 (March 1973).

Note that hashing techniques do not, in general, preserve order. Hence, if a large collection of records is to be retrieved and sorted, the collection itself should be ordered (e.g., in a sequential file) and only the index (or indexes) should be in a hashed-address table.

Generalized Models and Summary of Examples

This subsection has defined various alternative physical access paths, in order to define commonly used structures, to illustrate some of the complexities which can be otained from combinations of a few simple ideas, and to motivate the following subsections. The last three examples - TRIE-TREE, search trees, and hashing - present different compromises among retrieval and maintenance speed, storage efficiency, and reorganization. Hashing with open overflow can be very efficient, but is quite sensitive to loading factor and patterns in the data - periodic reorganization is necessary to avoid long sequences of synonyms, given a reasonably high loading factor. Chained overflow is less sensitive to loading factor, but chains caused by data patterns may be less localized on peripheral storage, and so require more accesses. The TRIE-TREE combination behaves much like hashing with chained overflow - e.g., long chains in the TREE may require reorganization of either TRIE or TREE. The search tree is not particularly outstanding in any respect, but does have the advantage that reorganization can be continual, thus providing upper bounds on retrieval times.

Severance[20,31] describes a generalized structure for character string searches, which consists of a one-level prefix search with a TRIE, a one-level prefix search with a TREE, and a search of a sequence of synonyms. Five parameters completely characterize a structure: 1) the number of

characters used in the TRIE search (if 0, then of course there is no TRIE search), 2) a Boolean variable indicating whether the result of the TRIE search is a data or structure record, 3) the number of characters, if any, in the TREE search, 4) the fraction of chained nodes in the TREE, and 5) the fraction of chained records in the final set of synonyms. Refer to the example of the set of colors, discussed earlier under TRIE and TREE. Various combinations of parameter values may be chosen to model many common access paths (search trees and hashing with open overflow do not, however, seem to be representable in Severance's model). For example, if the number of characters in the TREE search is equal to the total number of characters in the string, then the result is either direct addressing or addressing through an index, depending on whether the second parameter indicates mapping into data or structure. If the TRIE maps into a sequence of records or pointers, and there is no TREE, then the result is an inverted file. If there is no TRIE search, but there is a TREE search, then the result is an indexed sequential file. If there is neither TRIE nor TREE search, then the result is an array, a list, or an array with overflow, according to whether the fraction of chained synonyms is zero, one, or an intermediate value. If the TRIE involves some but not all of the total number of characters, there is no TREE, and if the synonyms are chained, then the result is division hashing with chained overflow.

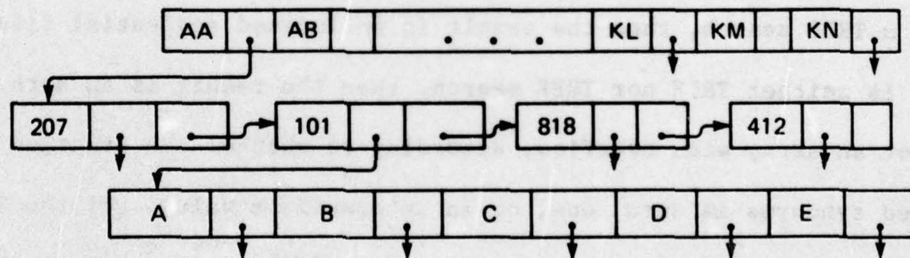
Severance[20,58] has developed a computer program which, given characteristics of hardware (speeds and sizes), data (sizes, volatility, and

---

58. Severance, D. G., "A Simulation Model for Basic File Organizations, ISDOS Working Paper #54", ISDOS Project, Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor (March, 1972).

usage), and design objectives (relative costs of space, maintenance time, and retrieval time), will generate and evaluate combinations of parameters and produce a near-optimal data base design.

Yao[21] describes a somewhat more general model, which allows for an arbitrary number of levels of TRIEs and TREEs. Yao has constructed a program, based on this model, for generating and evaluating near-optimal data base designs. Comparisons with other models suggest that Yao's model, cost equations, and optimization techniques are extremely accurate. Like Severance's model, Yao's model does not seem able to represent search trees and hashing with open overflow. Both are based on a TRIE-TREE sequence, which is not always appropriate. For example, if parts are identified by a string of two letters (the actually occurring letters are approximately consecutive, and hence dense), *followed by three numbers* (sparse), followed by one letter to denote sub-parts (consecutive), then the best organization is a TRIE-TREE-TRIE, as follows:



Either one three-letter TRIE or two consecutive TRIEs, followed by a TREE would be inefficient in storage, assuming that the number of subparts depends on the part. Assuming that sub-parts are frequently required consecutively, either would also require more time and peripheral accesses. A three-number, one-letter TREE would be inefficient in time, since the length of the TREE chains would be long. Neither optimization program is

able to exploit non-uniform record activity, nor to assign costs to in-core buffers, nor to deal with more than two storage levels (e.g., neither can model staging from archival to on-line storage). However, both programs are great improvements over the educated guesswork of traditional file design. Unfortunately, there are no published figures on the relative costs and accuracies of Severance's and Yao's models.

PARAMETERS OF PHYSICAL ACCESS PATHS

The subsection on Classifying Physical Access Paths described the basic concepts of access path design: data vs. structure, contiguity vs. chaining, ordering vs. no ordering, and direct vs. sequential access. As the previous subsection on examples illustrates, even fairly simple combinations of blocks can lead to the necessity for making a large number of decisions, many of which are by no means obvious. This section presents a further classification of these decision points.

Block Structure

The basic block structure must, of course, be determined. Organizations at each level, such as TRIE and hashing, TREE, and various search trees, were discussed above.

Free Space

The allocation, utilization, and recovery of free space is itself a problem of designing physical access paths. Free space may be located in one overflow area, as in ISAM, or it may be distributed throughout a number of levels of blocks, as in IBM's Virtual Storage Access Method[59]

---

59. Keehn, D. G., and J. O. Lacy, "VSAM Data Set Design Parameters", IBM Systems J., Vol. 13, No. 3, pp. 186-212 (1974).

(Martin, 1975, pp. 283-290).[19] Instead of a physical block overflowing, as it does in ISAM, it is split and roughly one-half is moved into a new area of free space; the old and new areas are then each about one-half full. The index is updated, which may cause another level of overflow. Space occupied by deleted records is immediately recovered. Insertion and deletion are therefore more time-consuming than in ISAM, but retrieval is faster, since overflow chains do not build up, and reorganization need not be performed as frequently. In VSAM, as in hashing with open overflow, the loading factor is quite important - too little free space increases insertion time, while too much is expensive in space and time since records are spread over more area.

Record Design

Record size and possible segmentation are clearly of great importance, particularly since the suitability of some physical access paths (especially hashing with open overflow) is critically dependent on record size. The previous section discussed some alternatives.

Hierarchical Storage

Different segments of a record, or records with different access frequencies, may be placed on different levels of hierarchical storage. Lum et al[60] present a cost analysis and various examples of allocation of a data base to different levels. Their results could be combined with the results of Eisner and Severance[15] on record segmentation.

---

60. Lum, V. Y., M. E. Senko, C. P. Wang, and H. Ling, "A Cost Oriented Algorithm for Data Set Allocation in Storage Hierarchies", CACM, Vol. 18, No. 6, pp. 318-322 (June 1975).

UTILIZATION PARAMETERS

Some requirements which must be met by a data base, and relevant design decisions, are described in the following paragraphs:

Rapid Response to Simple On-Line Queries

Accesses to peripheral storage should be minimized. Adherence to a user-oriented order is not particularly important, since output records can be quickly sorted, if necessary, during report generation. Direct access to the data, or a very simple index structure, is necessary. Hence a shallow block structure, contiguity, and order are required.

Rapid Response to Complex On-Line Queries

Direct access to the data is generally unacceptable, since too many large records would be examined. A deep block structure can be used to subdivide the data base into relevant areas to be searched exhaustively, or multiple indexes (such as inverted file or bounded multilist indexes) can be used so that query resolution can be performed on pointers instead of records. Indexes should be ordered and contiguous to permit direct access.

High-Volume Off-Line Retrieval

If a single user-defined report order is required, then records should be contiguous in that order to reduce access time and avoid subsequent sorting (refer to the analysis of the multilist retrieval). If other users require different report orders, separate indexes of pointers may be maintained, and sorting can then be performed by constructing channel programs.

On-Line Maintenance

Insertion may either be done immediately or be deferred by means of a special block which accumulates insertions. Distributed free space facilitates insertion into ordered, contiguous blocks; many levels of blocks, each with free space, minimize the amount of re-ordering necessary in such insertions. Deletion should be done by logically marking deleted records, which are then physically replaced by other records only when their space is needed. Modification of an indexed field on a direct access device, or of any field on a tape, is a deletion followed by an insertion; modification of a non-indexed field on a direct-access device is a simple search followed by a rewrite in place. Deferral of maintenance is attractive, because response times are more uniform and maintenance generally more efficient. Insertions can be ordered to reduce physical accesses and movement of data. However, the insertion block must be accessed separately, and in itself presents a problem in access path design.

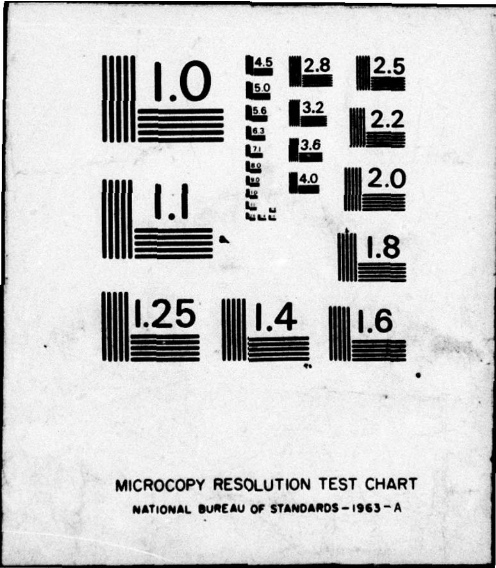
High-Volume Continuous Maintenance

Highly volatile files which require a user-defined order should be structured in many block levels with distributed free storage to minimize movement of data. Records should be chained rather than contiguous. Chains should be maintained separately rather than embedded in the data. If necessary, embedded chains should be two-way to simplify linking and unlinking. If possible, the order of records should be determined by the availability of free space. Data base keys and a translation table should be used rather than physical addresses, to reduce index updating if data records are moved.

AD-A035 945 DAVID W TAYLOR NAVAL SHIP RESEARCH AND DEVELOPMENT CE--ETC F/G 9/2
DATA BASE DESIGN.(U)
JUN 76 D K JEFFERSON
UNCLASSIFIED DTNSRDC-76-0111 NL

2 OF 2
AD A035945

END
DATE FILMED
3-77


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Storage

Storage costs can be reduced by eliminating redundant records, by allocating infrequently used blocks of a data base to less expensive devices, and by compression techniques, as in the preceding section. Pointer size can be reduced by the use of data base keys, which can be much shorter than physical addresses, and a translation table. This is particularly significant in an inverted file. Note that compression techniques can be used very effectively on indexes.

Loading and Reorganization

The efficiency of such structures as search trees and hashing tables can be very high if the distribution of record usage is highly skewed and loading is in descending order of record usage. Frequent reorganization tends to reduce the adverse effects of maintenance activity on relatively static structures with simple free-space structures, such as ISAM or hashing with open overflow. Such structures are also relatively easy to reorganize.

SUMMARY OF PHYSICAL ACCESS PATH DESIGN

This section presented the basic concepts with which one can design physical access paths that provide reasonable compromises among competing objectives, such as retrieval speed and storage cost. Commonly used examples were presented to illustrate the application of the concepts. Various design and evaluation models were cited. Finally, very rough generalizations were made concerning the attainment of selected specific objectives.

SUMMARY

The objective of this report has been to divide the broad subject of data base design into four separate areas - logical record design, logical access path design, physical record design, and physical access path design - and to provide a more or less independent analysis of each area. Logical record design has been shown to be a consequence of the real-world relationships being modelled. Logical access path design has been discussed as a choice among competing models - hierarchical, network, and relational - which all offer advantages and disadvantages. Physical record design has been primarily oriented toward reduction of storage costs. Record compression and segmentation have been discussed. Physical access path design has been the subject of the most lengthy and least decisive discussion. Objectives such as retrieval speed and storage efficiency generally conflict, and the number of possible combinations of basic elements is so large that only broad generalizations have been made. Numerous design and evaluation models have been cited.

REFERENCES

1. Codd, E. F., "Normalized Data Base Structure: A Brief Tutorial," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM, New York, pp. 1-17 (1971).

2. Codd, E. F., "Relational Completeness of Data Base Sublanguages," Proc. Courant Computer Science Symposium 6: Data Base Systems, Prentice-Hall, Englewood Cliffs, New Jersey, pp. 72-74 (1972).

3. Schmid, H. A. and R. J. Swenson, "On the Semantics of the Relational Data Model," Proc. 1975 ACM-SIGMOD International Conference on the Management of Data, ACM, New York, pp. 211-223 (1975).

4. Bernstein, P. A., J. R. Swenson, and D. C. Tsichritzis, "A Unified Approach to Functional Dependencies and Relations," Proc. 1975 ACM-SIGMOD International Conference on the Management of Data, ACM, New York, pp. 237-245 (1975).

5. Boyce, R. F., and D. D. Chamberlin, "Using a Structured English Query Language as a Data Definition Facility," IBM Research Laboratory, San Jose, California (December 10, 1973).

6. Chamberlin, D. D., J. N. Gray, A. L. Traiger, "Views, Authorization and Locking in a Relational Data Base Design," IBM Research Laboratory, San Jose, California (October 7, 1974).

7. Date, C. J., "An Introduction to Database Systems", Addison-Wesley, Reading, Mass. (1975), pp. 137-221.

8. CODASYL, "CODASYL Data Base Task Group Report, April 1971", ACM, New York (1971).

9. CODASYL, "CODASYL Data Description Language, Journal of Development, June 1973," National Bureau of Standards, Washington, D.C. (1973).

10. Codd, E. F. "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM, New York, pp. 35-68 (1971).

11. Chamberlin, D. D. and R. F. Boyce, "SEQUEL: A Structured English Query Lanuage," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM, New York (1974).

12. Lorie, R. A., "XRM - An Extended (N-ary) Relational Memory," Technical Report 320-2096, IBM Scientific Center, Cambridge, Mass. (January, 1974).

13. Fano, R. M., "Transmission of Information", M.I.T. Press, Cambridge, Mass. (1961).

14. Huffman, D. A., "A Method for the Construction of Minimum-Redundancy Codes," Proc. I.R.E., Vol. 40, p. 1098 (September, 1952).

15. Eisner, M. J., and D. G. Severance, "Mathematical Techniques for Efficient Record Segmentation in Large Shared Data Bases", Technical Report No. 261, Department of Operations Research, Cornell University, Ithaca, New York (July, 1975).

16. Bobrow, D. G., "A Note on Hash Linking", CACM, Vol. 18, No. 7, pp. 413-415 (July, 1975).

17. Martin, J., "Computer Data-Base Organization," Prentice Hall, Englewood Cliffs, New Jersey, p. 397 (1975).

18. IBM Corporation, "Introduction to IBM Direct-Acess Storage Devices and Organization Methods, GC20-1694-8", IBM, White Plains, New York, pp. 55-63 (1974).

19. Knuth, D. E., "The Art of Computer Programming, Volume 3: Sorting and Searching", Addison-Wesley, Reading, Massachusetts, pp.396-399 (1973).

20. Severance, D. G., "Some Generalized Modeling Structures for Use in Design of File Organizations", PhD dissertation, the University of Michigan, Ann Arbor (1972).

21. Yao, S. B., "Evaluation and Optimization of File Organizations Through Analytic Modeling", Ph.D. dissertation, The University of Michigan, Ann Arbor (1974).

22. Smith, S. E., and J. H. Mommens, "Automatic Generation of Physical Data Base Structures", Proc. 1975 ACM-SIGMOD International Conference on the Management of Data, ACM, New York, pp. 157-165 (1975).

23. Gerritsen, R., "A Preliminary System for the Design of DBTG Data Structures", The Wharton School, University of Pennsylvania, Philadelphia, (1975). Also published in CACM, Vol. 18, No. 10, pp. 551-557 (October, 1975).

24. Senko, M. E., E. B. Altman, M. M. Astrahan, and P. L. Fehder, "Data Structures and Accessing in Data-Base Systems", IBM Systems J., Vol. 12, No. 1, pp. 30-93 (1973).

25. Schneider, L. S. and C. R. Spath, "Quantitative Data Description", Proc. 1975 ACM-SIGMOD International Conference on the Management of Data, ACM, New York, pp. 169-185 (1975).

26. Cardenas, A. F., "Evaluation and Selection of File Organization - A Model and System", CACM, Vol. 16, No. 9, pp. 540-548 (September, 1973).

27. Lefkovitz, D., "File Structures for On-Line Systems", Hayden, Rochelle Park, New Jersey, pp. 126-129, 157-165 (1969).

28. Lefkovitz, D., "Data Management for On-Line Systems", Hayden, Rochelle Park, New Jersey, pp. 62-64, 118-120 (1974).

29. Fredkin, E., "TRIE Memory", CACM, Vol. 3, pp. 490-499 (September 1960).

30. de la Briandais, R., "File Searching Using Variable Length Keys", Proc. 1959 Western Joint Computer Conference, IEEE, New York, pp. 295-298 (1959).

31. Severance, D. G., "Identifier Search Mechanisms: A Survey and General Model", ACM Computing Surveys, Vol. 6, No. 3, p. 186 (September 1974).

32. Nievergelt, J., "Binary Search Trees and File Organization", ACM Computing Surveys, Vol. 6, No. 3, pp. 195-207 (September 1974).

33. Foster, C. C., "A Generalization of AVL Trees", CACM, Vol. 16, No. 8, pp. 513-517 (August 1973).

34. Bayer, R. "Binary B-trees for Virtual Memory", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM, New York, pp. 219-235 (1971).

35. Bayer, R., "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", Acta Informatica, Vol. 1, No. 4, pp. 290-306 (1972).

36. Bayer, R., and E. M. McCreight, "Organization and Maintenance of Large Ordered Indexes", Acta Informatica, Vol. 1, No. 3, pp. 173-189 (1972).

37. Nievergelt, J., and E. M. Reingold, "Binary Search Trees of Bounded Balance", SIAM J. Computing, Vol. 2, No. 1, pp. 33-43 (March 1973).

38. Held, G., and M. Stonebraker, "B-trees Re-examined", Memorandum No. ERL-M528, Electronics Research Laboratory, University of California, Berkeley (2 July 1975).

39. Maurer, W. D., and T. G. Lewis, "Hash Table Methods", ACM Computing Surveys, Vol. 7, No. 1, pp. 6-19 (March 1975).

40. Peterson, W. W., "Addressing for Random-Access Storage", IBM J. Research and Development, Vol. 1, No. 2, pp. 130-146 (April 1975).

41. Lum, V. Y., P. S. T. Yuen, and M. Dodd, "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files", CACM, Vol. 14, No. 4, pp. 228-239 (April 1971).

42. Lum, V. Y.,and P. S. T. Yuen, "Additional Results on Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files", CACM, Vol. 5, No. 11, pp. 996-997 (November 1972).

43. Buchholz, W., "File Organization and Addressing", IBM Systems J., Vol. 2, pp. 86-111 (June 1963).

44. Maurer, W. D., "An Improved Hash Code for Scatter Storage", CACM, Vol. 11, No. 1, pp. 35-38 (January 1968).

45. Radke, C. E., "The Use of Quadratic Residue Research", CACM, Vol. 13, No. 2, pp. 103-105 (February 1970).

46. Day, A. C., "Full Table Quadratic Quotient Searching", CACM, Vol. 13, No. 8, pp. 481-482 (August 1970).

47. Batagelj, V., "The Quadratic Hash Method When the Table Size Is Not a Prime Number", CACM, Vol. 18, No. 4, pp. 216-217 (April 1975).

48. Ackerman, A. F., "Quadratic Search for Hash Tables of Size p^n", CACM, Vol. 17, No. 3, p. 164 (March 1974).

49. Hopgood, F. R. A., and J. Davenport, "The Quadratic Hash Method When the Table Size Is a Power of 2", Computer J., Vol. 15, No. 4, pp. 314-315 (1972).

50. Bell, J. R., and C. H. Kaman, "The Linear Quotient Hash Code", CACM, Vol. 13, No. 11, pp. 675-677 (November 1970).

51. Lamport, L., "Comment on Bell's Quadratic Quotient Method for Hash Code Searching", CACM, Vol. 13, No. 9, pp. 573-574 (September 1970).

52. Bell, J. R., "Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering", CACM, Vol. 13, No. 2, pp. 107-109 (February 1970).

53. Burkhard, W. A., "Full Table Quadratic Quotient Searching", Computer J., Vol. 18, No. 2, pp. 161-163 (1975).

54. Severance, D. G., and R. Duhne, "A Practitioner's Guide to Addressing Algorithms: A Collection of Reference Tables and Rules of Thumb", Technical Report No. 240, Department of Operations Research, Cornell University, Ithaca, New York (November 1974).

55. van der Pool, J. A., "Optimum Storage Allocation for Initial Loading of a File", IBM J. Research and Development, Vol. 16, No. 6, pp. 579-586 (November 1972).

56. van der Pool, J. A., "Optimum Storage Allocation for a File in Steady State", IBM J. Research and Development, Vol. 17, No. 1, pp. 27-38 (January 1973).

57. van der Pool, J. A., "Optimum Storage Allocationn for a File with Open Addressing", IBM J. Research and Development, Vol. 17, No. 2 (March 1973).

58. Severance, D. G., "A Simulation Model for Basic File Organizations, ISDOS Working Paper #54", ISDOS Project, Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor (March, 1972).

59. Keehn, D. G., and J. O. Lacy, "VSAM Data Set Design Parameters", IBM Systems J., Vol. 13, No. 3, pp. 186-212 (1974).

60. Lum, V. Y., M. E. Senko, C. P. Wang, and H. Ling, "A Cost Oriented Algorithm for Data Set Allocation in Storage Hierarchies", CACM, Vol. 18, No. 6, pp. 318-322 (June 1975).

INITIAL DISTRIBUTION

3 CNO

- 1 OP91
- 1 OP911F
- 1 OP911G

1 CHONR 437

1 ARPA/INFO PROC TECH

2 NRL

- 1 5400
- 1 5403

1 NAVPGSCOL 53

1 NADC 504

2 NSWC/Dahlgren Lab

- 1 DK70
- 1 DK74

1 NELC 5200

1 NAVELEX 330

2 NAVSUP

- 1 0413
- 1 0431

1 NAVSEA 03C

2 NAVAIR

- 1 503Z
- 1 4013E

1 NFMSO 964

1 NAVSEC 6105B

12 DDC

CENTER DISTRIBUTION

Copies	Code	Copies	Code
1	18	1	186
1	1805	30	188
1	1806	1	189
1	1809.3	30	5214.1
1	183	1	5221
1	185	1	5222